

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A238 160



DTIC
ELECTE
JUL 15 1991
S B D

DISSERTATION

A TYPE CALCULUS FOR MATHEMATICAL
PROGRAMMING MODELING LANGUAGES

by

Robert D. Clemence, Jr.

September, 1990

Dissertation Supervisor:

Gordon H. Bradley

Approved for public release; distribution is unlimited

91-04963



91 7 12 092

k. ADDRESS (City, State, and ZIP Code)

10. SOURCE OF FUNDING NUMBERS

PROGRAM
ELEMENT NO.

PROJECT
NO.

TASK
NO.

WORK UNIT
ACCESSION NO

1. TITLE (Include Security Classification)

A TYPE CALCULUS FOR MATHEMATICAL PROGRAMMING MODELING LANGUAGES

2. PERSONAL AUTHOR(S)

Clemence, Robert D. Jr.

3a. TYPE OF REPORT
Dissertation

13b. TIME COVERED
FROM _ TO _

14. DATE OF REPORT (Year, Month, Day)
September 1990

15. PAGE COUNT
143

6. SUPPLEMENTARY NOTATION

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

7. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

data types, integrated modeling, linear programming,
model validation, mathematical programming software,
special purpose languages

9. ABSTRACT (Continue on reverse if necessary and identify by block number)

The issue of model validation is critical in the formulation and interpretation of mathematical programming models, yet this problem is largely ignored by contemporary modeling languages and the systems they support. This research advances modeling languages for mathematical programming by providing a formalism and defining a language for specifying a dimensional complement, called "typing," to the algebraic representation of models. Typing is a formal specification used to determine automatically whether the algebraic model is well-formed in the sense that its objective function and constraints are composed of homogeneous components and that operations performed using indices are meaningful. A provision is made for the definition of dimensional axioms that can be applied automatically to resolve dimensional differences. The addition of formal typing to mathematical programming models also yields a powerful abstraction mechanism for integrated modeling.

Approved for public release; distribution unlimited.

A Type Calculus for Mathematical Programming
Modeling Languages

by

Robert D. Clemence, Jr.
Major, United States Army
B.S., Lehigh University, 1973
M.S., Naval Postgraduate School, 1984

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN OPERATIONS RESEARCH

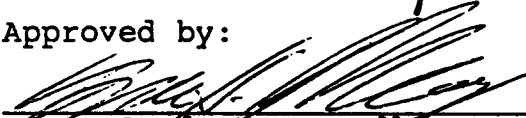
from the

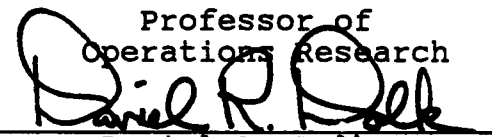
NAVAL POSTGRADUATE SCHOOL
September 1990

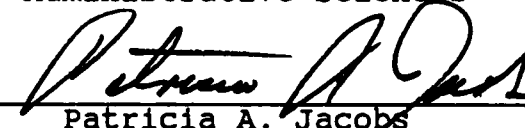
Author:

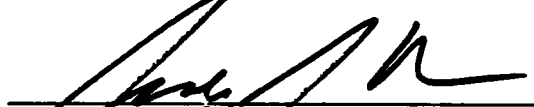

Robert D. Clemence, Jr.

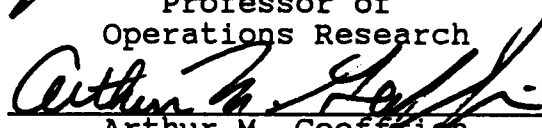
Approved by:



Gordon H. Bradley
Dissertation Advisor
Professor of
Operations Research


Daniel R. Dolk
Associate Professor
Administrative Sciences



Patricia A. Jacobs
Professor of
Operations Research


Gerald G. Brown
Chairman
Professor of
Operations Research

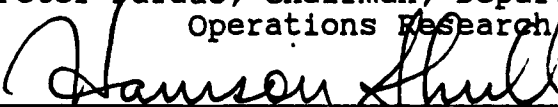

Arthur M. Geoffrion
Professor
The John E. Anderson Graduate
School of Management at UCLA


R. Kevin Wood
Associate Professor of
Operations Research

Approved by:


Peter Purdue, Chairman, Department of
Operations Research

Approved by:


Harrison Shull, Provost/Academic Dean

ABSTRACT

The issue of model validation is critical in the formulation and interpretation of mathematical programming models, yet this problem is largely ignored by contemporary modeling languages and the systems they support. This research advances modeling languages for mathematical programming by providing a formalism and defining a language for specifying a dimensional complement, called "typing," to the algebraic representation of models. Typing is a formal specification used to determine automatically whether the algebraic model is well-formed in the sense that its objective function and constraints are composed of homogeneous components and that operations performed using indices are meaningful. A provision is made for the definition of dimensional axioms that can be applied automatically to resolve dimensional differences. The addition of formal typing to mathematical programming models also yields a powerful abstraction mechanism for integrated modeling.



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION	1
II.	FUNDAMENTALS OF TYPING	6
	A. DATA TYPES IN PROGRAMMING LANGUAGES	6
	B. STARTING WITH BASICS: UNITS OF MEASUREMENT	7
	C. CONTINUING: DIMENSIONAL CHARACTERISTICS	9
	D. ATTRIBUTING DIMENSIONAL CHARACTERISTICS TO ENTITIES AND EVENTS: CONCEPTS	12
	E. A GLOSSARY FOR TYPING NUMERICAL OBJECTS	12
	F. TYPING INDEX USAGE	13
	G. EXAMPLE	16
III.	PRINCIPLES: ADDING TYPING TO A MODELING LANGUAGE	20
	A. HOW MODELING LANGUAGE SYSTEMS FOR LINEAR PROGRAMMING WORK	20
	B. HOW A TYPE LANGUAGE SYSTEM WOULD WORK	22
	C. HOW A MODELING LANGUAGE SYSTEM AND A TYPE LANGUAGE SYSTEM WOULD WORK TOGETHER	25
IV.	A TYPE LANGUAGE	29
	A. NUMERICAL TYPE SYNTACTIC STRUCTURE	29
	1. Operators.....	30
	2. Reserve Words	30
	3. Constants and Symbolic Names	32
	4. Quantity Declarations and Concept Declarations	33
	5. Dimensional and Unit Components	33
	6. Numerical Type and Index Type Statements	35

B.	NUMERICAL TYPE SEMANTICS	35
1.	Arithmetic, Assignment and Relational Operators	35
2.	Determination of Equivalence	37
3.	Type Conversions	47
4.	Type Coercions	51
C.	INDEX TYPE SYNTAX AND SEMANTICS	64
1.	Fundamentals	64
2.	Index Operators	64
3.	Set Operators	67
V.	LANGUAGE EXTENSIONS	69
A.	POLYMORPHIC TYPES	69
B.	CONCEPT GRAPHS	72
C.	INDEXED CONCEPTS	78
D.	APPLICATION DOMAINS	85
VI.	TYPING APPLIED TO INTEGRATED MODELING	89
A.	INTEGRATED MODELING	89
1.	A Rationale For Building Models of Systems As Integrated Models	89
2.	The Mechanics of Integrating Algebraic Models	91
B.	INTEGRATING ALGEBRAIC MODELS WITH UNTYPED AND TYPED MODELING LANGUAGES	92
1.	Integrated Modeling in an Untyped Language	92
2.	Integrated Modeling in a Typed Language	106
C.	INTEGRATED MODELING WITH LIBRARY UNITS	116
1.	An Introductory Example	116
2.	Library Units	119

3. Integrated Modeling With Library Units	122
VII. CONCLUSION	125
APPENDIX A: A PARTIAL GRAMMAR FOR AN ELEMENTARY MODELING LANGUAGE	127
LIST OF REFERENCES	129
BIBLIOGRAPHY	132
INITIAL DISTRIBUTION LIST	133

ACKNOWLEDGEMENT

This work would not have been possible without the endurance and support of my wife, Marilyn, and the patience of my three children: Dale, Robert and Elizabeth. The pages that follow will, I hope, answer a question I was asked daily for five years: "What did you do in school today, Daddy?"

I. INTRODUCTION

During the past 30 years, parallel developments in optimization technology and computer technology have greatly increased the size and complexity of solvable mathematical programming models. As the size and complexity of solvable models increase, more general and effective support tools are needed to enable formulation efforts to keep pace with optimization.

Historically, considerable attention has been focused on one aspect of modeling support: the translation of a modeler's algebraic formulation into the computational data structures required by a solution algorithm. This task involves the substitution of real and integer numbers for symbolic parameters in objective functions and constraints. Typically, the result of this procedure is a compact representation of a very sparse matrix whose rows and columns number in the hundreds or thousands and whose non-zero elements appear in intricate patterns. The size and complexity of such a structure makes manual translation impractical.

Recognition of the need for a specialized language that supports modeler's algebraic notation has led to the development of *modeling languages* (ML) for mathematical programming [e.g., Bisschop and Meeraus <1982>, Burger <1982>, Clemence <1984>, Fourer, Gay and Kernighan <1987>, Geoffrion <1988>, and Lucas and Mitra <1988>]. Modeling languages are declarative programming languages designed to emulate the algebraic notation used by modelers to express mathematical programming models. They provide constructs for representing parameters, variables, functions and constraints. Modeling languages designed for formulating large-scale problems also allow these constructs to be defined over sets with multiple indices. Because there is nearly a one-to-one relationship between a modeler's personal notation and the features of the language,

creation of a modeling language program or *schema*¹ is more a task of transcription than translation for the modeler.

Given a schema and data for the schema, modeling language systems produce a file that is ready for solution by an optimization system. Each particular data set, together with its corresponding schema, forms an instance of the model that we will call a "problem." When the schema and the data are stored in separate files, the model user, who need not be the same person as the modeler, has the power to formulate many different problems by combining the schema with different data files. In situations where the model changes as much as, or more than the data, schema and data can be combined in a single file. Whether the schema and data are separate or intermixed, it is useful to view the execution of a problem by a system as two processes:

- (1) an algebraic validation that determines if objects are correctly defined and if sets, indices, functions, and constraints are composed with valid operations; and
- (2) a data validation that determines if all the data are present and in the form required by the model.

In contemporary modeling language systems, the algebraic validation and data validation are relatively weak. Most detectable errors are typographical and are easily corrected. The real substance of the validation is contained in the name and the explanatory description associated with each numerically valued symbol in the model. These descriptions are vital because they enable the purely numerical results obtained from solving the algebraic representation to be interpreted in real-world terms.

¹The word "schema" is used by Geoffrion <1988> to describe models composed as Structured Models. We prefer this term to "modeling language program" because formulation in a modeling language is not the same as creating a matrix generator in a computer language like FORTRAN.

To ensure that the algebraic form of his model correctly captures his intention, the modeler is obliged to perform a "dimensional" check of each algebraic function and constraint he specifies. This is done by replacing each numerically valued symbol by its explanatory description and then applying two kinds of dimensional calculus. One kind is the calculus of measurement units: a unit analysis must be performed to verify that pure numbers that are added, subtracted or compared have the same scale of reference. The other calculus is similar in intent to the first: it verifies that pure numbers that are added, subtracted or compared either represent the same real world phenomena or can be made the same by applying some rule or abstraction. For example, suppose " X " represents the weight of *apples* measured in pounds and " Y " represents the weight of *oranges* measured in pounds. If the expression " $X + Y$ " were to appear in a constraint, the modeler might resolve the difference in the descriptions of " X " and " Y " by assigning their sum the description "weight of *fruit* measured in pounds."

Since modeling languages have been primarily designed for accessing pure numbers from data files and storing them in the data structures required for computation, few facilities have been provided for describing what the symbols mean. Documentation of the meaning of a variable or a parameter is limited, in most languages, to the use of meaningful names and in-line commentary that is not processed. For example, GAMS [Bisschop and Meeraus <1982>] and LEXICON [Clemence <1984>] require an interpretation as part of the declaration of a symbol, but the style and content of that description is still a matter of personal taste. Because these descriptions are informal, verification of the modeler's intention must still be done as a separate, manual exercise.

In this thesis we develop a formalism, called *typing*, for automating dimensional checking. Typing describes the meaning of parameters, variables, functions and constraints, and the numerical characteristics of indices. In our paradigm, the modeler

formulates an algebraic model in a modeling language and provides an explanatory description in a type language for each of the aforementioned constructs in the schema. The computer then processes this extended schema and automatically verifies that both representations are in agreement before it creates computational data structures.

Typing makes models more secure. When the modeler's intention is expressed as an executable complement to the model schema, the two representations are tightly coupled. A change in one that is inconsistent with the other can be automatically detected and brought to the attention of the modeler or model user. Not only does typing provide a facility for defining and enforcing dimensional consistency (a feature totally lacking in existing modeling languages for mathematical programming), it also yields a powerful abstraction mechanism for creating templates of models and for integrated modeling.

The specification of constructs and notation for what have previously been informal ideas must be done carefully. An emphasis has been placed in our research on the design of a type calculus that is general enough to encompass all existing algebraic modeling languages for mathematical programming. In our examples, typing is added to a "generic" modeling language, referred to as "EML" (Elementary Modeling Language) in the sequel, that contains the principal features of several systems [Bisschop and Meeraus <1982>, Clemence <1984>, Fourer <1983>, and Fourer, Gay and Kernighan <1987>]. EML supports multiple indexing, allows data to be endogenous or exogenous to the schema and permits parameters to be defined as functions. A meta-language specification of EML is provided as Appendix A.

Our research is presented in six chapters. The first three chapters develop the typing formalism and define a notation and grammar for its implementation. Chapter II describes the fundamentals of typing modeling language parameters, variables, constraints, functions and index sets. Chapter III describes how typing would be added to a modeling language

system for linear programming. Chapter IV presents the syntax and semantics of a language kernel based on this paradigm. The language kernel is then extended in Chapter V with four more features: polymorphic types, type indexing, a structure to facilitate type coercions and a mechanism for standardizing and encapsulating types for particular applications. Chapter VI discusses the utility of typing to integrated modeling. Chapter VII presents our research conclusions.

II. FUNDAMENTALS OF TYPING

We begin this chapter by reviewing the use of data types in programming languages. Typing of numerically-valued objects in modeling languages will then be introduced incrementally. Units of measurement will be discussed first, followed by dimensional characteristics (classical concerns of dimensional analysis, e.g., Bridgeman <1935>) and then the idea of concepts. Typing of index sets in modeling languages will be discussed in Section C. The chapter concludes with an example of a typed modeling language schema.

A. DATA TYPES IN PROGRAMMING LANGUAGES

Programmers make errors. Although the developers of programming methodologies seek to prevent errors at the source, the nature of the task is such that programming errors will not be eliminated entirely. It is therefore useful to assist the programmer in detecting, identifying and correcting them. Data types can be effective tools in this endeavor.

A data type consists of a set of values and a collection of operations defined over that set. The values of a data type establish a convention on how the contents of an address in computer memory should be interpreted by the host machine. Specification of allowable operations on these values forestalls certain programming errors by preventing meaningless operations from being performed. Thus, data types increase the security of computer programming by preventing violations of the type structure of a language from going undetected.

A distinguishing feature of languages for mathematical modeling is the provision of specialized data types such as variables, parameters, functions and equations in addition to the integer, real and logical types needed for scientific computation. These specialized types can also be indexed over sets of finite, discrete elements.

Data types in contemporary modeling languages are not secure. While sets may only be manipulated with other sets, operations on set elements which are order-dependent can be applied to any set, whether its elements are ordered or unordered. Similarly, variables, parameters and functions are of equal status and may be added, subtracted, multiplied, divided and compared with impunity. In the sequel, we will develop criteria for guaranteeing the security of mathematical operations on these constructs.

B. STARTING WITH BASICS: UNITS OF MEASUREMENT

Numbers by themselves have arbitrary meaning. Even labelled numbers are still ambiguous, albeit that labels can suggest what a number may represent. For example, "*steel* := 6" is plausibly more meaningful than "*x* := 6". To be meaningful, a number must be associated with a *unit of measurement*. A unit of measurement is a standard of comparison used to ascertain the extent of something. If we say "*x*" is measured in tons, we know implicitly that any number assigned to "*x*" is a denominate number: it represents the number of times that 1 ton occurs in the thing that "*x*" represents. Furthermore, "*x*" should not be added, subtracted, or compared with "*y*" unless both "*x*" and "*y*" are expressed in tons.

The responsibility for the security of arithmetic computations in programming languages is traditionally split between people and computers. Since computer arithmetic emulates the mathematics of the real number line, programming languages disassociate numerical values from units of measurement. Numbers without dimensions can be compared and combined as elements of the real number system without risk. The programmer is solely responsible for performing a complementary computation in terms of a measurement system to verify the units of his arithmetic expressions. Modeling languages have perpetuated this division of labor as the direct descendants of general programming languages.

The task of specifying units of measurement for each symbolic numerical value in a program and manually performing unit analysis is tedious and error-prone. These characteristics and the fact that unit computation is performed according to fixed rules makes unit analysis an obvious candidate for automation. Thus, one characteristic of a modeling language data type should be unit of measurement.

The idea that numerical data objects should always be associated with units of measurement has been attributed to Hoare <1973>. With this information, the language compiler could check the validity of proposed operations beyond mere numerical feasibility. For example, a value determined by dividing a value in "miles" by a value in "gallons" should only be assigned to a data object measured in "miles/gallon". The inclusion of a unit of measurement along with a numerical value in a data type has the benefits of increasing the reliability and readability of the calculations as well as increasing the security of the program itself. Proposals for languages with units of measurement have been made by Gehani <1977>, Karr and Loveman <1977>, and House <1983>. A working implementation of Hoare's idea has been actualized only recently as an extension to the PASCAL programming language by Dreiheller, Moerschbacker and Mohr <1986>. However, as we demonstrate in the following example, units alone are insufficient to make modeling language arithmetic calculations secure.

At a superficial level, extending the type structure of a modeling language to include units of measurement is simple. Begin by declaring a unit for each variable and parameter in the ML representation. Then, program a rule for each of the arithmetic operations performed on denominate numbers. For example,

Division results in the ratio of the units of measure of the two operands being inherited by the quotient. Identical units in numerator and denominator of a unit description cancel one another.

C. CONTINUING: DIMENSIONAL CHARACTERISTICS

A rule-driven approach based on units of measurement alone would be naive. Consider an excerpt from the hypothetical formulation of a capital-budgeting investment problem as a mathematical program (Figure 2.1).

Objective Function:

$$\text{Max } \sum_j c_j x_j + \sum_i P_i (\min(0, b_i - \sum_j a_{ij} x_j)) + \dots$$

where:

j = a set of investment alternatives

i = an investment year

c_j = a present discounted value in year 1 of an income stream of investment j (\$)

b_i = available budget in year i to invest (\$)

P_i = the penalty parameter for violating the available budget in year i

x_j = binary variable indicating whether or not alternative j is selected

a_{ij} = the capital outlay required by investment j in year i

Figure 2.1 Model Excerpt

Should P_i be considered to be a dimensional or a dimensionless parameter? One way to determine the answer would be to form the ratio of the units of " $c_j x_j$ " and " $b_i - \sum_j a_{ij} x_j$."

Since both terms are expressed in dollars, we might conclude that P_i must be dimensionless: no unit conversion is necessary to make the two terms conformable. The subtlety missed by a rule based only on units is that present worth expressed in dollars, and

a lump sum in a future time period expressed in dollars are not the same. Hence, P_i must have the units

$$\frac{\text{present worth (dollars)}}{\text{violated budget (dollars) in year } i}$$

Unit of measurement alone is insufficient to convey information accurately to someone else. Scientific observation requires two kinds of descriptions: a quantitative description so that the observed phenomenon can be distinguished from other phenomena; and a unit of measurement to distinguish quantitatively similar occurrences of different magnitude.

A quantitative description is a description based upon the conventions of a dimensional system. A dimensional system is a set of fundamental quantities together with a set of rules for determining all other quantities in the system from this fundamental quantity set. In physics and engineering, standards for quantitative description are established. Quantitative information about physical systems or events is described in terms of products of fundamental quantities, such as force, length or time, raised to appropriate powers. For example, the quantity "area" can be described as "length²." Table 2.1 provides descriptions of certain engineering quantities in terms of force (F), length (L) and time (T).

Although force, length, and time are regarded as the fundamental dimensions for engineering problems, many other combinations could be considered fundamental. In physics, mass (M) is considered more fundamental than force. Force, mass, length and time are interrelated through Newton's second law of motion,

$$\text{force} = \text{mass} * \text{acceleration} .$$

Stated in dimensional form, $F = ML/T^2$.

TABLE 2.1 ENGINEERING QUANTITIES AND MEASURES

Quantity	FLT Basis	English	Metric
Force	F	lb	nwt
Mass	FT^2/L	lb-sec ² /ft	kg = nwt · sec ² /m
Length	L	ft	m
Area	L^2	ft ²	m ²
Volume	L^3	ft ³	m ³
Velocity	L/T	ft/sec	m/sec
Acceleration	L/T^2	ft/sec ²	m/sec ²
Pressure	F/L^2	lb/ft ²	nwt/m ²
Energy	FL	ft · lb	joule = nwt · m

This table lists the terms and identities used by engineers as an example of a standard for dimensional description.

In general, any quantity may be expressed as the product of fundamental quantities raised to appropriate powers. The unit of measurement employed in a scientific observation is a standard within some measurement system. A measurement system is created by establishing a unit of measurement for each fundamental quantity of a dimensional system and determining all other units of measurement from adopted dimensional conventions. English and metric standard units for selected engineering quantities are provided in Table 2.1.

Inherent in our linear system of measurement and idea of scientific observation is the postulate that numerical values of the same magnitude are not equal unless they describe the same quantity and are derived from identical units.² Consequently, the operations of

²Dimensional analysis, a technique employed to obtain information about the form of a solution to a physical problem, is based upon the principle of dimensional homogeneity of physical equations. That is, every term in a complete and general physical equation must have the same dimension, expressed in fundamental quantities. (e.g., Bridgeman <1935>).

addition and subtraction, and the use of the relational operators ($\leq, =, \geq$) are only meaningful when their operands can be reduced to a common quantitative and measurement standard.

D. ATTRIBUTING DIMENSIONAL CHARACTERISTICS TO ENTITIES AND EVENTS: CONCEPTS

Numerically-valued objects are included in models to represent a quantifiable behavior of an entity or the occurrence/non-occurrence of an event considered important in the modeled problem. We call this entity/event a *concept* and consider it to be an essential part of any modeling description. A concept is an abstraction intended to summarize the common characteristics of the particulars it subsumes. Each concept has an associated set of measurable quantities. For example, the concept "rectangle" has the quantities "length", "width", and "height." An important distinction between concepts and quantities is that concepts do not have exponents. For example, "width" of "rectangle" multiplied by itself would be "width²" of "rectangle," not "width²" of "rectangle²."

E. A GLOSSARY FOR TYPING NUMERICAL OBJECTS

We now define a data type for numerically valued symbols in ML schemas. A *numerical type* is a data type that has two components, a *dimensional description* and a *unit description*. The dimensional description is an executable description of the phenomena represented by a modeler-named object or an arithmetic expression of model-named objects. It has two parts: a *quantity* and a *concept* possessing that quantity. Each concept has an associated set of measurable quantities. Our choice of the term "quantity" is intended to be analogous to our earlier use of the same word when describing conventions for scientific observation. Weight, value, cardinality, and duration are examples of quantities.

A unit description is composed of a *unit of measurement* and an optional *scale factor*. A unit of measurement is the standard of comparison to be applied when arithmetic,

assignment and comparison operations are performed between two numerically typed operands. It can be composed of a product form or ratio of product forms of fundamental units defined in the ML schema by the modeler. For example, if "meter" and "kilogram" are declared as fundamental units, then "meter²" and "kilogram / meter³" are allowable units of measurement. A scale factor is an unsigned multiplier that amplifies or diminishes a unit of measure. For example, "100 meters²" and "1/10 kilograms / meter³" are unit descriptions that include scale factors.

Each symbol or combination of symbols and arithmetic operators capable of having a numerical value is assigned a numerical type. Type assignment is performed by the modeler for data objects (parameters, variables, functions and constraints) he names. He does this by writing a *type declaration* in a *type language* for each named data object in the ML schema. The lexical and syntactical details of the type language are deferred until Chapter 4. Types of arithmetic expressions composed of these named objects and numerical literals (e.g., 2, 1.23, .10E-7) are determined by a *type analyzer*. The type analyzer is a computer program that parses the type language, interprets its expressions, and converts them to a lower-level form for *type verification* and *data verification*. Type verification is the process of determining the consistency of arithmetic expressions according to the rules of composition, or *calculus*, of the type language. Data verification is the process of determining whether the explicitly-typed data submitted as values for an ML schema's typed names match the numerical types expected in that context. The syntax and semantics of a numerical type language are presented in Chapter 4.

F. TYPING INDEX USAGE

Symbolic indexing enhances the conciseness, precision and generality of algebraic modeling notation. It provides an ability to group collections of symbols and an ability to

manipulate those collections using elementary set operations and logical operators. For example, statements like

$$\sum_{(i,j) \in S} X_{ij}$$

make it possible to include or exclude particular objects or subsets of objects in functionals and constraints.

The advantages of indexed notation derive in part from an assumption of homogeneity. When a modeler chooses to group objects into a set, say "X," and distinguish between them by an index, say "j," an assertion is made that some uniform conditions hold for each set element that can be abstracted in the form of a typical element, say "X_j." If all X_j are homogeneous with respect to their characteristics (except, perhaps, their numerical value), then the whole set of X-objects can be described in a single modeling language definition. If the X_j are not uniform, and the distinctions are important, the modeler has two alternatives: (1) partition the X_j elements into named, homogeneous subsets of objects; or (2) provide individual definitions for each element of the set.

The rest of the advantages of indexed notation result from the numerical characteristics that modelers ascribe to the indices they use. Although modelers tend to blur the distinction, there are three kinds of simple indices (as opposed to compound index sets whose elements are tuples of simple indices) used in modelers' personal notations. *Nominal indices* represent domains of unordered labels. When a nominal index is appended to the label of a set of data objects, it is similar to distinguishing the members of a family by their first names. Since nominal indices are unordered, the only relationships defined between nominal indices are "equal" and "not equal."

The second kind of simple index is the *ordinal index*. Ordinal indices represent domains of simply ordered labels. In addition to the "equal" and "not equal" relationships,

ordinal index labels can be compared using the "greater than" and "less than" relations to other index labels within their domains.

The third type of simple index is the *ordinal+ index*. Sets assigned ordinal+ indices are simply ordered. In addition, each element of an ordinal+ index set acquires the integer value associated with its ordinal position, counting from one to the cardinality of the set. Hence, ordinal+ indices inherit not only the relations and operators used with nominal and ordinal indices, but certain integer arithmetic operations as well.

We believe the distinctions between the three kinds of simple indices are significant enough to justify index typing in modeling languages. The use of ordinal indices is already considered important in programming languages. Enumerated types, ordered collection of identifiers, are included in PASCAL (e.g., Jensen and Wirth (1983)) to circumvent the abuse of integer data types in index usage. For example, in FORTRAN IV (e.g., McCracken <1965>) an integer in the closed interval [1,7] can be used to represent a day of the week. This same coded object can then be raised to a rational power later in the program without violating the syntax or semantics of FORTRAN IV.

Typing indices in modeling language increases model security by regulating the ways in which indices can be used as selectors in iterated arithmetic operations and as operands in arithmetic computation. This classification also reminds the modeler/model user of the model's sensitivity to index set modifications. If, for example, an index is declared to be "nominal", the user is assured that the associated index set can be enlarged, reduced or permuted without affecting the properties of the model. Permutation of the elements of an "ordinal+" index set, however, incurs a risk of changing the intended outcome of iterated arithmetic operations that depend on the original order. Consider an example where a modeler wishes to sum the diagonal elements of an $n \times n$ dimension table which is indexed by two "ordinal+" sets. The row index of the table is "i." The column index of the table is

"j." The index set "SET" provides an (i,j) address for each cell in table "X." Let "Position()" be an operator that converts the ordinal position of an index set element into an integer. A modeling language statement that accomplishes this task is

SUM (I,J) {SET} [POSITION(I) EQ POSITION(J)](X(I,J)).

If the elements of either index set are rearranged, the outcome of this operation will not be what the modeler intended.

We sometimes find it convenient in modeling to create compound index sets composed of k-tuples of simple indices. For example, tuples of nominal index values like (RENO,CHICAGO) may be defined in a transportation model to designate the beginning and ending points of a route. An example of convolved index types is (DALLAS, JULY_87) where "DALLAS" is a nominal index representing a location and "JULY_87" is an element of an ordinal+ index set which represents the months in a three-year period. When a simple index becomes a component in a compound index it maintains the properties and operators associated with its index type. The syntax and semantics of an index type language are presented in Chapter 4.

G. EXAMPLE

Figure 2.2 displays a linear programming model written in EML and a computer language designed for typing. Typing descriptions are enclosed in double angle brackets,

e.g., << nominal >> ,

to aid explanation now, and implementation later. This notation makes it easier for the reader and a computer implementation of these notations to discern typing constructs from ML constructs. In addition, keeping ML statements separate from typing statements increases the portability of typed ML schemas. If the brackets are replaced by the symbols used in an ML to enclose in-line comments, typed ML schemas can be ignored on modeling systems that do not support typing.

Two kinds of statements occur in almost all computer languages. Declaratives are used to declare facts that are needed before a program can be executed, such as the symbolic names of areas of memory and the initial contents of memory areas. Imperatives are commands that are executed during the running of a program, such as assignment and computational operations. Four kinds of type language declaratives are used in Figure 2.2. The statements indented beneath the headings "QUANTITIES" and "CONCEPTS" establish what we call a *numerical type context* for the model. A *quantity statement* declares the existence of a particular quantity and associates it with a pre-defined system of measurement. For example, the statement "WEIGHT : LBS" asserts that a quantity called WEIGHT is an atom in the model's type structure and that the modeler intends WEIGHT to be measured in pounds. A *concept statement* attributes a set of quantities to a particular concept. The statement "@BUTTER[WEIGHT]" declares the existence of a concept called @BUTTER that has WEIGHT as a measurable quantity. Quantities are declared prior to concepts in accordance with a well-known programming language design principle (e.g., Wiener and Sincovec <1982>) which we will refer to as *Define Before Use*. For our purposes, "define before use" means

All objects named by the programmer should be declared prior to their use in other declarative and imperative statements.

Adherence to this principle facilitates model understanding by others by requiring the modeler to present his formulation in a non-circuitous way. It also simplifies the design of a language compiler by allowing a schema to be processed in one pass (e.g., Aho and Ullman <1977>).

```

<< QUANTITIES
  WEIGHT : LBS ;
  COST : US_$ ;
  DURATION: DAY ; >>

<< CONCEPTS
  @BUTTER [WEIGHT] ;
  @OBJECTIVE [COST] ;
  @TIME [DURATION] ; >>

SETS
  DAIRIES i; << nominal >>
  WAREHOUSES j; << nominal >>
  PATHS(i,j) := { CROSS ({DAIRIES} , {WAREHOUSES}) };

VARIABLES

  SHIPMENT(i,j) {PATHS}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>
  POSITIVE: SHIPMENT(i,j);

PARAMETERS
  SCOST(i,j) {PATHS}; << COST of @OBJECTIVE / (WEIGHT of @BUTTER
                                     / DURATION of @TIME) # US_$ / ( [100] LBS / DAY ) # >>

  SUPPLY(i) {DAIRIES}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

  DEMAND(j) {WAREHOUSES}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

FUNCTIONS
  OBJECTIVE := SUM (i,j) {PATHS} (SCOST(i,j)*SHIPMENT(i,j));
                                     << COST of @OBJECTIVE # US_$ # >>

CONSTRAINTS
  OUTBOUND(i) {DAIRIES} := SUM (j) {PATHS} (SHIPMENT(i,j)) =L= SUPPLY(i);
                                     << WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY # >>

  INBOUND(j) {WAREHOUSES} := SUM (i) {WAREHOUSES} (SHIPMENT(i,j)) =E= DEMAND(j);
                                     << WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY # >>

SOLVE
  MIN OBJECTIVE; SUBJECT TO ALL;

REPORT
  SHIPMENT(i,j) {PATHS}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

```

Figure 2.2 EML Schema With Typing

The concept, quantity and unit primitives declared in the type context are used to define the third kind of declarative, the *numerical type statement*. Numerical type statements are used to declare types for each parameter, variable, function and constraint named by the modeler in the EML schema. The EML variable SHIPMENT(i,j) has the type statement

WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY #.

The first part of the statement is called the *dimensional description*. It describes the phenomena that FLOW(i,j) represents in the EML schema. The second part of the type statement, enclosed in #..#, is called the *unit description*. It associates the numerical value of SHIPMENT(i,j) with a unit of measurement. In this example, the unit LBS is prefaced by a scale factor of "100" to indicate that SHIPMENT(i,j) is measured in units of 100 pounds per day.

The last kind of declarative statement used in Figure 4.1 is the *index type statement*. Each index set in the EML schema that introduces an index, such as DAIRIES i, has one. The index type statement restricts permissible transformations on index set elements, such as lag or lead operations.

A language for typing has no imperatives of its own; instead, it uses the arithmetic statements of an ML. When a type language is executed, the type of each parameter, variable, function and index is substituted for its symbolic name in functions and constraints. These expressions are then evaluated according to the semantics of the type language. Operations defined on indices, such as "equal to" (e.g., (i EQ j), are imperatives for the evaluation of index types.

III. PRINCIPLES: ADDING TYPING TO A MODELING LANGUAGE

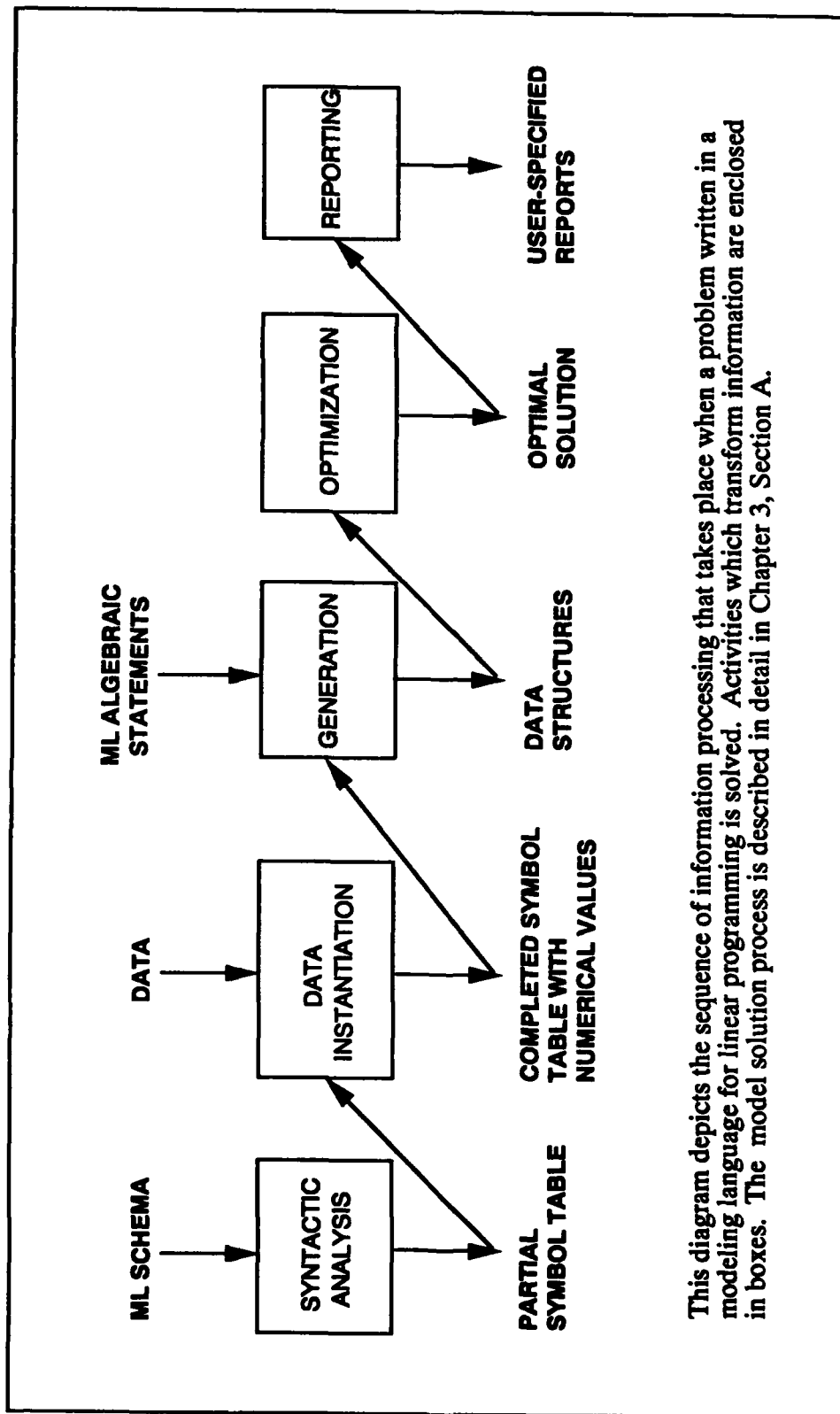
A type language and its type analyzer are the complement of a modeling language and its modeling language translator. The type language portion of a model provides a formal specification that can be used to determine whether modeling language algebraic expressions are well-formed in a more restricted sense than in the arithmetic of real or integer numbers. The purpose of this chapter is to describe how these two processes would work together in a modeling language for linear programming.

A. HOW MODELING LANGUAGE SYSTEMS FOR LINEAR PROGRAMMING WORK

Figure 3.1 is a diagram of the process of transforming a linear program, expressed in modeling language, into optimal solution results.³ The process has five stages. During *syntactic analysis*, a modeling language translator performs two functions: it parses each modeling language statement and ensures its consistency with statements that precede it; and, it constructs a symbol table. This symbol table is used to access the character strings of index values and the numerical values of parameters, variables and functions.

The second stage of the process is *instantiation*. At this point, the generic algebraic model is transformed into a concrete model instance by specifying the elements of index sets and by assigning numerical values to model parameters. After this data is extracted from the model text file, or from a separate source, a modeling language translator performs checks for completeness. Every index set must have at least one element and every parameter must have a value.

³This is the method employed by Clemence <1984> in the "Lexicon" ML. Individual MLs may vary in some details.



This diagram depicts the sequence of information processing that takes place when a problem written in a modeling language for linear programming is solved. Activities which transform information are enclosed in boxes. The model solution process is described in detail in Chapter 3, Section A.

Figure 3.1 Model Solution Process

Once the data requirements of the model are satisfied, the third stage of the process, *generation*, can begin. During generation, sets of algebraic functions and constraints are translated in a four-step procedure into vectors of real numbers. Translation is done in order of appearance in the model schema. Lexicographical order is followed within sets.

The first step in the generation procedure is called *reduction*. During reduction, each function or constraint is reduced to an algebraic expression composed of variables, parameters, symbolic numerical literals and arithmetic operators. This is done by repeatedly replacing each occurrence of a function name by its algebraic definition and by replacing iterated operators by their primitive forms.

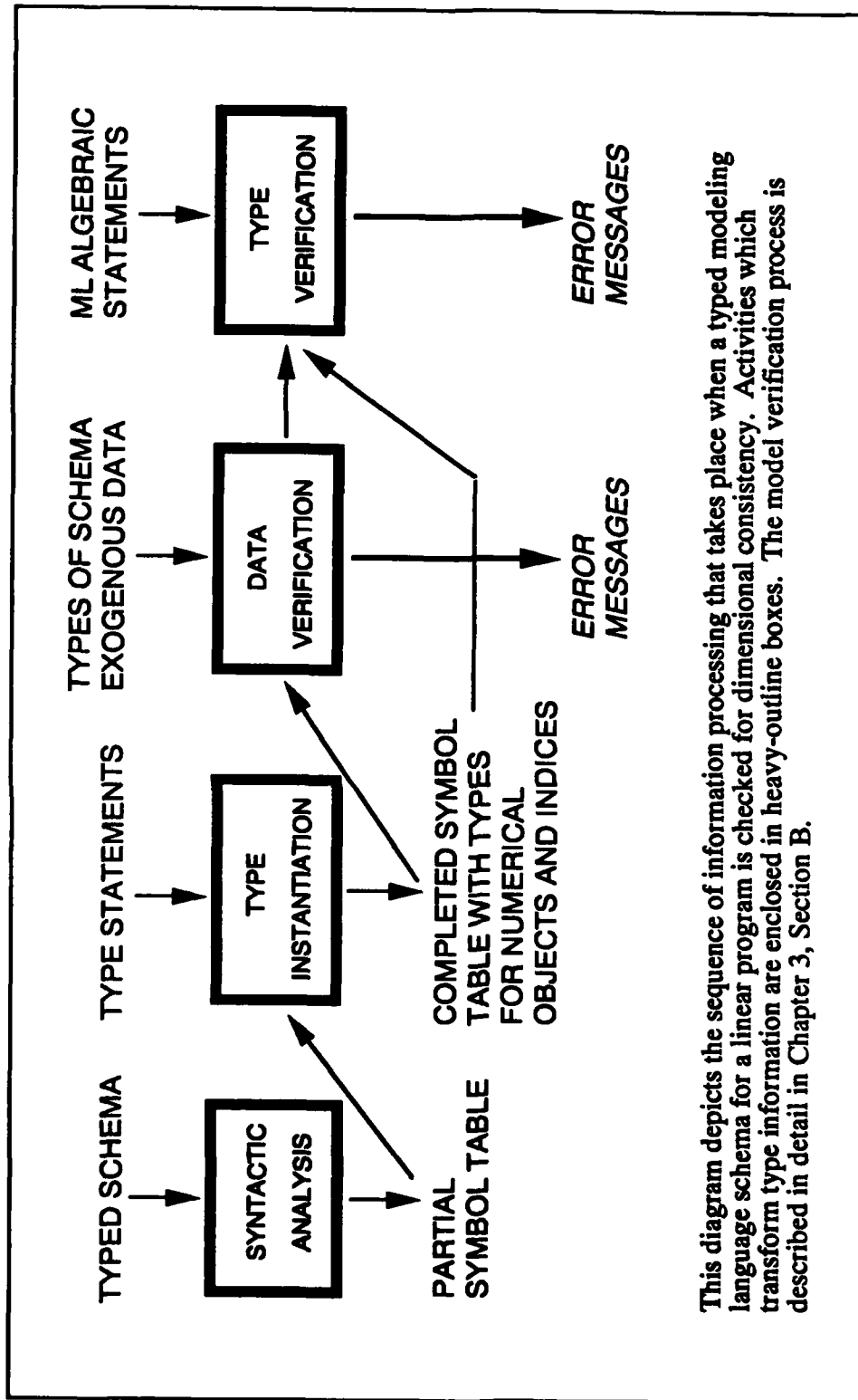
After reduction is complete, the next generation step is *encoding*. The algebraic *infix* notation used by the modeler to specify functions and constraints, i.e., $\langle \text{operand} \rangle \langle \text{operator} \rangle \langle \text{operand} \rangle$, is converted to an encoded *postfix* notation, i.e., $\langle \text{operand} \rangle \langle \text{operand} \rangle \langle \text{operator} \rangle$, for manipulation on a push-down data structure (e.g., Aho, Hopcroft and Ullman <1974>).

The last step in the generation stage, *evaluation*, has two responsibilities. It extracts numerical values from the symbol table and it evaluates the postfix coding to produce either a scalar, an objective function vector or a row of the constraint matrix.

The last two stages in Figure 3.1 are *optimization* and *report generation*. Optimization includes the tasks of invoking the solution algorithm, storing optimized results, and computing auxiliary functions that require those results. Report generation is self-explanatory.

B. HOW A TYPE LANGUAGE SYSTEM WOULD WORK

Determination of whether modeling language functions and constraints are consistent in terms of the typing system is a four-stage process (Figure 3.2). These stages are *syntactic analysis*, *type instantiation*, *data verification*, and a counterpart to generation,



This diagram depicts the sequence of information processing that takes place when a typed modeling language schema for a linear program is checked for dimensional consistency. Activities which transform type information are enclosed in heavy-outline boxes. The model verification process is described in detail in Chapter 3, Section B.

Figure 3.2 Model Verification Process

called *type verification*. The functions performed during syntactic analysis by a type analyzer are the same as those performed by a modeling language translator: parsing, syntactic checking, and symbol table construction. An important difference between type language syntactic analysis and modeling language syntactic analysis is that a type analyzer must be able to parse and manipulate modeling language as well as its own language. This is necessary in order to associate types with modeling language identifiers and to perform necessary functions in the type verification stage.

The second stage in the process, type instantiation, assigns a numerical type or an index type to each operand in the model schema. This information may be included within the schema or specified as model data.

The third stage in the process is data verification. When the numerical data that specifies a model instance is provided by an external source, there is a risk that these numbers will not mean what the modeler expected they would mean. If this data is typed, a type analyzer can compare the description provided by the source with the modeler's original specification. Data values that do not meet specification can be brought to the attention of a modeling language translator and the modeler.

When each index and each parameter, function, and variable in the model has an assigned type, the type verification stage can be performed. Like matrix generation, it is a four-stage process: it reduces modeling language functions and constraints to simpler, equivalent forms composed of only variables, parameters, numerical literals and arithmetic operators; it converts these equivalent forms to an encoded postfix notation; and it evaluates the encoded form using a push-down data structure.

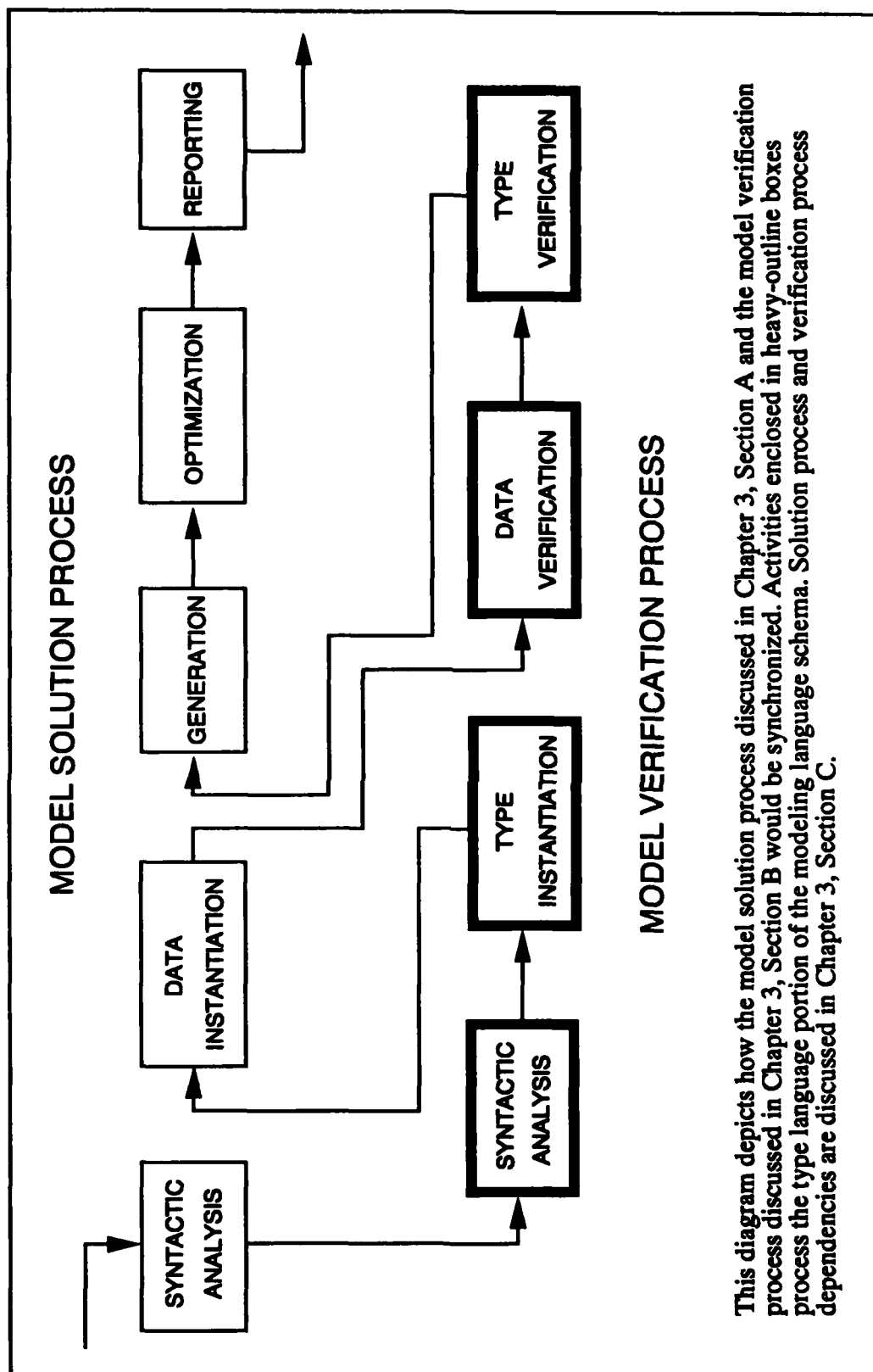
Although matrix generation and type verification recognize the same arithmetic operator symbols and obey the same rules of operator precedence when processing modeling language functions and constraints, neither the semantics of those operations nor

their results are the same. Matrix generation manipulates numbers. There is a one-to-many mapping between each modeling language function and constraint in a schema, and the functions and constraints of a problem instance. The number of functions and constraints generated from the schema for each instance is determined by the problem data. The output of the matrix generation stage is a vector of coefficients intended for a solver. Type evaluation manipulates symbols. Each modeling language function and constraint is evaluated only once. The output of the type verification stage, for each function and constraint, is an indicator intended for a modeling language translator and the model user. The indicator signifies whether a particular function or constraint is "safe" to generate as a scalar or a vector of numerical constants. The additional work needed to do this kind of checking is independent of problem size.

C. HOW A MODELING LANGUAGE SYSTEM AND A TYPE LANGUAGE SYSTEM WOULD WORK TOGETHER

Figure 3.3 is a diagram that depicts the synchronization of the two processes. Notice that the syntactic analysis performed by a type analyzer occurs after that of a modeling language translator. This sequencing simplifies the job of a type analyzer considerably since it can assume that all modeling language statements it processes are free of errors in syntax. Data verification has been scheduled after data instantiation arbitrarily. It would also be possible to reverse this ordering and use data verification as a data preprocessor. The type verification stage of the typing process occurs after the model instance is specified but before the requisite computational data is generated. This is done to interdict the production of type-inconsistent functions and constraints.

If the purview of a type language is limited to that of a "go - no go" gauge for exogenous model data and for matrix generation, a type analyzer and a modeling language translator may operate semi-autonomously. The only couplings between the two systems



This diagram depicts how the model solution process discussed in Chapter 3, Section A and the model verification process discussed in Chapter 3, Section B would be synchronized. Activities enclosed in heavy-outline boxes process the type language portion of the modeling language schema. Solution process and verification process dependencies are discussed in Chapter 3, Section C.

Figure 3.3 Synchronization Of Model Generation And Model Verification Processes

are their sequencing, the ability of a type analyzer to recognize the modeling language grammar, and a channel that a type analyzer can use to inform a modeling language translator of its deductions.

Extension of a type language's range of control to include the ability to alter numerical values necessitates a tighter coupling between its type analyzer and a modeling language translator. If a type analyzer assumes the responsibility of converting the units of incoming data to a system of measurement specified by the modeler, it must be able to reproduce the numerical input format read by the modeling language translator. Alternately, it must be able to access numerical values through the symbol table of a modeling language translator and alter them in situ. This knowledge of the inner workings of a modeling language translator is essential to perform the symmetric extension: the ability to change the units of optimized results for reports.

The most powerful extension of a type language's numerical responsibility would be for it to autonomously perform unit conversions within modeling language functions and constraints. This enhancement is more difficult to implement than the tailoring of input or output because the effect of each transformation must be local in context, not global. If, for example, the units of a parameter are changed from "pounds" to "grams" to resolve a conflict of units in a sub-expression of a constraint, that does not imply that this change is necessary elsewhere in the model. Hence, globally changing the units of a numerical object by altering its assigned numerical value, as would be done for input or output, is prohibited.

The ability to perform unit transformations on a case-by-case basis necessitates the tightest coupling between a type analyzer and a modeling language translator. A type analyzer must have the ability to name conversion factors and to place them, and their numerical values, in a modeling language translator's symbol table and attendant data

structures. It must also have the ability to create revised forms of constraints and functions which include these conversion factors that the modeling language translator can process to generate scalars, objective function vectors and matrix rows. One way to accomplish this second task would be to employ a type analyzer's type verification stage as a preprocessor for a modeling language translator's generation stage. The linkage between the two systems would be the encoded postfix form required for arithmetic evaluation on a push-down data structure. This procedure is explained in detail in Chapter 4.

IV. A TYPE LANGUAGE

Each formal language has its own jargon and its own set of peculiar symbols and constructs; our type language is no exception. The purpose of this chapter is to introduce our terminology and symbology. The grammar of our numerical type language is formally presented in a programming meta-language in Section A. Numerical type semantics are specified in Section B. Grammar and semantics for index types are presented in Section C.

A. NUMERICAL TYPE SYNTACTIC STRUCTURE

In this section we present formal definitions for the lexical and syntactic conventions of the numerical type language. Programming languages are described in terms of grammars (e.g., Aho and Ullman <1977>). A grammar is a finite set of syntactic categories. Syntactic categories are defined recursively in terms of each other and primitive symbols called terminals. The terminals we use are elements of the ASCII⁴ character set (any standard computer-readable scheme would do equally well). The rules which relate syntactic categories to each other are called productions.

The grammar of the numerical type language grammar is described in the sequel using a meta-language known as extended Backus-Naur form (BNF) (e.g., MacLennan <1983>). BNF notation represents syntactic categories by words or phrases in angle brackets such as "< digit >." The symbol "::=" is read as "is defined as." Juxtaposition of syntactic categories means they are concatenated; spaces between symbols and syntactic categories represent blank characters. The symbol "|" is read as "or" and is used to list

⁴A(merican) S(tandard) C(ode) for I(nformation) I(nterchange): a proposed standard for defining codes for information exchange between computer equipment produced by different manufacturers.

alternate forms that the thing being described can take. The notation "<..>*" stands for a sequence of zero or more occurrences of a syntactic category. If the superscript "*" is replaced by a superscript "+" the minimal number of occurrences in the sequence is one instead of zero. The same superscripts can be used with parentheses to stand for multiple occurrences of the enclosed contents. Constructs contained in square brackets, "[..]", are optional. Terminals are enclosed in double quotes. These conventions are summarized in Table 4.1. Figure 4.1 is provided as an example of numerical type language constructs.

TABLE 4.1 EXTENDED BNF NOTATION

Symbol	Interpretation
< >	syntactic category
[]	optional
()	to be treated as a single term
	or
" "	terminal
::=	is defined as
superscript *	zero or more occurrences of
superscript +	one or more occurrences of

1 . Operators

The numerical type language includes the conventional arithmetic operators, parentheses, and a special operator for mapping a quantity to a system of measurement. The symbols chosen to represent these operators appear in Table 4.2.

2 . Reserve Words

Reserve words are words that a language reserves for its own use; they cannot be used by the programmer as labels. Figure 4.1 uses two language reserve words:

```

<< QUANTITIES
  WEIGHT : LBS ;
  COST : US_$ ;
  DURATION: DAY ; >>

<< CONCEPTS
  @BUTTER [WEIGHT] ;
  @OBJECTIVE [COST] ;
  @TIME [DURATION] ; >>

SETS
  DAIRIES i; << nominal >>
  WAREHOUSES j; << nominal >>
  PATHS(i,j) := { CROSS ({DAIRIES} , {WAREHOUSES}) };

VARIABLES

  SHIPMENT(i,j) {PATHS}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>
  POSITIVE: SHIPMENT(i,j);

PARAMETERS
  SCOST(i,j) {PATHS}; << COST of @OBJECTIVE / (WEIGHT of @BUTTER
                                     / DURATION of @TIME) # US_$ / ([100] LBS / DAY) # >>

  SUPPLY(i) {DAIRIES}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

  DEMAND(j) {WAREHOUSES}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

FUNCTIONS
  OBJECTIVE := SUM (i,j) {PATHS} (SCOST(i,j)*SHIPMENT(i,j));
                                     << COST of @OBJECTIVE # US_$ # >>

CONSTRAINTS
  OUTBOUND(i) {DAIRIES} := SUM (j) {PATHS} (SHIPMENT(i,j)) =L= SUPPLY(i);
                                     << WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY # >>

  INBOUND(j) {WAREHOUSES} := SUM (i) {WAREHOUSES} (SHIPMENT(i,j)) =E= DEMAND(j);
                                     << WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY # >>

SOLVE
  MIN OBJECTIVE; SUBJECT TO ALL;

REPORT
  SHIPMENT(i,j) {PATHS}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

```

Figure 4.1 EML Schema With Typing

TABLE 4.2 OPERATORS

Symbol	Interpretation
+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation
()	parentheses
:	unit designator

The symbols listed in this table are primitives that will be used to define the syntax and semantics of our type language

"QUANTITIES" and "CONCEPTS". Other reserve words are declared in the sequel as new language features are introduced. Details on the use of QUANTITIES and CCNCEPTS are provided later.

3 . Constants and Symbolic Names

Extended BNF definitions for numerical constants are given in Figure 4.2. The language recognizes three forms of numbers: integers, real numbers expressed in decimal or scientific form and real numbers expressed in rational form. Whereas decimal and scientific forms are implemented as floating point numbers, rationals have their own implementation. They are used in the language in situations were floating point representation could create ambiguity, such as when scale factors are compared for equality.

Three kinds of symbolic names are used to build more complex forms: unit labels, quantity labels and concept labels (See Figure 4.3). Concept labels are distinguished from unit and quantity labels by their "@" character prefix.


```

<digit> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
<unsigned integer> ::= <digit>+
<signed integer> ::= ["+"|"-"] <unsigned integer>
<rational> ::= ["+"|"-"] <unsigned integer> "/" <unsigned integer>
               |<unsigned integer>
<decimal number> ::= <unsigned integer> "." <unsigned integer>
<scientific number> ::= <decimal number> "E" ["-"|"+"] <unsigned integer>

```

Figure 4.2 Numerical Constants

```

<uppercase letter> ::= "A"|"B"|"C"|...|"Z"
<special character> ::= "_"|"$"|"%"
<alphanumeric> ::= <uppercase letter>|<digit>|<special character>
<label> ::= <uppercase letter><alphanumeric>*
<unit label> ::= <label>
<quantity label> ::= <label>
<concept label> ::= "@"<label>

```

Figure 4.3 Identifiers

4. Quantity Declarations and Concept Declarations

Quantity and concept declarations establish the type context for all numerical types used in a typed model schema. Each quantity statement in a quantity declaration introduces a unique quantity label and asserts how that quantity label will be measured. Each concept statement in a concept declaration associates quantity labels with a unique concept label. The syntax for these constructs is given in Figure 4.4.

5. Dimensional and Unit Components

Dimensional components and unit components are intermediate constructs used to form numerical types. Each is defined recursively. This is done to show that new numerical types can be created dynamically when arithmetic operations are evaluated. The

syntax for dimensional components and unit components is given in Figure 4.5. The purpose of the universal dimensional component and universal unit component is discussed

```

<quantity statement> ::= <quantity label> ":" <unit label>

<quantity declaration> ::= "QUANTITIES" <quantity statement>
                           (";" <quantity statement>)* ";"

<concept statement> ::= <concept label> "[" <quantity label>
                       (, <quantity label>)* "]"

<concept declaration> ::= "CONCEPTS" <concept statement>
                          (";" <concept statements>)* ";"

```

Figure 4.4 Concept and Quality Declarations

```

<quantity> ::= <quantity label>
              | <quantity> "^" <unsigned integer> <quantity> "*" <quantity>
              | <quantity> "/" <quantity> | "(" <quantity> ")"

<universal dimensional component> ::= "@*"

<dimensional clause> ::= <quantity> "of" <concept label>
                       | <universal dimensional component>

<dimensional component> ::= <dimensional clause>
                           | <dimensional component> "*" <dimensional component>
                           | <dimensional component> "/" <dimensional component>
                           | (<dimensional component>)

<scale factor> ::= "[" <rational> "]"

<universal unit component> ::= "UNITY"

<unit> ::= <unit label>
          | <unit> "^" <unsigned integer> <unit> "*" <unit> | <unit> "/" <unit>
          | "(" <unit> ")"

<unit clause> ::= [<scale factor>] <unit> | <universal unit component>

<unit component> ::= <unit clause>
                    | <unit component> "*" <unit component>
                    | <unit component> "/" <unit component>
                    | "(" <unit component> ")"
                    | <universal unit component>

```

Figure 4.5 Dimensional and Unit Components

at length later in this chapter. Both constructs are used to create modeler-defined numerical type and unit transformations.

6. Numerical Type Statement

BNF definition for the numerical type statement is given in Figure 4.6.

<numerical type statement> ::= <dimensional component> "#" <unit component> "#"

Figure 4.6 Numerical Type Statement

B. NUMERICAL TYPE SEMANTICS

During the first two stages of model verification, the type analyzer parses declarative statements to establish a collection of labels for defining numerical types and index types, and stores the types specified for modeler-named EML objects (index sets, variables, parameters, functions and constraints). The lexical and syntactic conventions of declarative statements were described in the previous section. During the last two stages of model verification, the type analyzer applies a type calculus to determine whether EML arithmetic and assignment imperative statements are well-formed. The purpose of this section is to present the principles and rules for manipulating numerical types. We begin by describing the semantics of numerical type arithmetic, numerical type assignment and numerical type comparison. We conclude by discussing three data typing notions: equivalence, conversion and coercion.

1. Arithmetic, Assignment and Relational Operations

During the execution phase, each function and constraint defined in an EML schema is checked to determine its *numerical type validity*. A function or constraint is *numerically type valid* if the computed numerical type of its algebraic expression is the same as the one declared for it by the modeler.

How does one determine the numerical type of an algebraic expression? Any symbolic numerical value (parameter name, variable name, or function name) or numerical literal (9, 3.14159, 0.10E-7) is an arithmetic expression by itself. In this base case, simply replace the label by its modeler declared numerical type. Numerical types for longer algebraic expressions are determined in a way analogous to computing a numerical result. A numerical result is computed by replacing each symbolic numerical value by its numerical value and then applying arithmetic operators according to their precedence. For example, in EML, the "+" operator takes real numbers as operands and returns a real number. A numerical type result is determined by replacing each symbolic numerical value by its modeler-declared or previously computed type and then applying arithmetic operators according to their EML precedence. In this context, the "+" operator, and all other arithmetic operators, take types as operands and return types as resultants. Whereas numerical arithmetic is well-understood, arithmetic on numerical types requires explanation.

Numerical type arithmetic is based on three principles. First, the product or ratio of valid types is always a valid type. Second, for the resultant of type addition or type subtraction to be a valid type, both operands must be the same type. If this condition is not met, the sum or difference is a distinguished type called a *numerical type error*. Third, the resultant type of any arithmetic operation involving an operand of numerical type error is a numerical type error.

In addition to the numerical type error, there is another distinguished type called the *universal type*. A universal type is a numerical type with a dimensional component of "@" and a unit component of "UNITY." Adding or subtracting the universal type from another valid numerical type is analogous to adding or subtracting zero from a real number. Multiplying or dividing a valid numerical type by a universal type is analogous to

multiplying or dividing a real number by one. In all cases, the real number and the valid numerical type are unchanged by the operation.

Universal types originate in three ways. First, a modeler can define a symbolic numerical value, such as a parameter, to have the universal type. Assigning universal types to all parameters, variables and functions has the effect of disabling the type verification mechanism: all arithmetic, assignment, and comparison operations are legal when operands have the universal type. Second, the product of numerical type multiplication or the quotient of numerical type division can have the universal type if the dimensional and unit components of the operands cancel one another. The third source of universal types is the existence of non-exponent numerical literals in the EML schema. These multipliers are assigned the universal type by the type analyzer during execution.

Numerical type arithmetic is performed by manipulating the quantities, concept labels, scale factors and unit labels associated with each EML operand according to semantic rules. These rules are detailed in four tables. Each table contains an EML grammar production for an arithmetic imperative and the rules used to determine its type. Numerical type addition and subtraction are defined in Table 4.3. Next, numerical type multiplication and division rules are defined in Tables 4.4 and 4.5, respectively. Finally, numerical type exponentiation is defined in Table 4.6. The notation used in these tables is as follows: $@A, @B$ are concept labels; a, b are quantities with scale factors s_a, s_b and unit labels u_a, u_b , respectively; E_1, E_2 are symbolic numerical values; R is the symbolic numerical resultant; and N is an unsigned integer.

2. Determination of Equivalence

The validity of real addition, real subtraction, assignment and comparison operations on symbolic numerical values is decided by whether or not the operands have the same type. The standard used to make this determination is a variant of one known in

TABLE 4.3 TYPE ADDITION AND SUBTRACTION

NOTATION KEY FOR TABLES 4.3, 4.4, 4.5 and 4.6	
Concepts	@A, @B
Quantities	a, b
Units	u _a , u _b
Scale Factors	s _a , s _b
Symbolic Numerical Operands	E ₁ , E ₂
Symbolic Numerical Resultant	R
Signed Integer	N
Production	Semantic Rule
R <== E ₁ + E ₂	R.type ::= IF E ₁ .type = a of @A # s _a u _a # AND E ₂ .type = a of @A # s _a u _a # THEN a of @A # s _a u _a # ELSE IF E ₁ .type = a of @A # s _a u _a # AND E ₂ .type = @* # UNITY # THEN a of @A # s _a u _a # ELSE <i>numerical type error</i>
R <== E ₁ - E ₂	
R <== -E ₁ + E ₂	

programming languages as *structural equivalence*. Under structural equivalence, two objects are considered to have the same type if the structural description of their types is the same. Consider this excerpt from Figure 4.1:

VARIABLE SHIPMENT(I,J) {PATH}; << WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY # >>

PARAMETER DEMAND(J) {WAREHOUSES}; << WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY # >>.

Do SHIPMENT(I,J) and DEMAND(J) have the same numerical type? According to the structural equivalence rule they do since both declarations are constructed from the same numerical type context primitives, using the same operators ("/", "*"), in the same sequence.

TABLE 4.4 TYPE MULTIPLICATION

Production	Semantic Rule
$R \Leftarrow E_1 * E_2$	<pre> R.type ::= IF E1.type = a of @A # s_a u_a# AND E2.type = b of @A # s_b u_b# THEN a * b of @A # s_a * s_b u_a* u_b # ELSE IF E1.type = a of @A # s_a u_a# AND E2.type = b of @B # s_b u_b# THEN a of @A * b of @B # s_a * s_b u_a* u_b# ELSE IF E1.type = a of @A # s_a u_a# AND E2.type = @* # UNITY # THEN a of @A # s_a u_a# ELSE <i>numerical type error</i> </pre>

TABLE 4.5 TYPE DIVISION

Production	Semantic Rule
$R \Leftarrow E_1 / E_2$	<pre> R.type ::= IF E1.type = a of @A # s_a u_a# AND E2.type = b of @A # s_b u_b# THEN a / b of @A # s_a / s_b u_a/ u_b # ELSE IF E1.type = a of @A # s_a u_a# AND E2.type = b of @B # s_b u_b# THEN a of @A / b of @B # s_a / s_b u_a/ u_b# ELSE IF E1.type = a of @A # s_a u_a# AND E2.type = @* # UNITY # THEN a of @A # s_a u_a# ELSE <i>numerical type error</i> </pre>

TABLE 4.6 TYPE EXPONENTIATION

Production	Semantic Rule
$R \Leftarrow E_1 \wedge E_2$	$R.type ::= \text{ IF } E_1.type = a \text{ of } @A \# s_a u_a \# \text{ AND } E_2.type = N$ $\text{ THEN } a \wedge N \text{ of } @A \# s_a \wedge N u_a \wedge N \#$ $\text{ ELSE } \textit{numerical type error}$

Although the structural equivalence rule is straightforward, it is counter-intuitive when applied to numerical types derived for longer arithmetic expressions. Multiplication, for example, is both commutative and associative in the field of real numbers. Under structural equivalence, two product forms of strictly equivalent numerical types would not be recognized as the same type if their operands were permuted. A solution to this dilemma is to prescribe that user-defined types and types derived during the evaluation of arithmetic expressions be placed in a canonical form before the structural equivalence rule is applied.

We now illustrate the construction and manipulation of a canonical form for numerical types through an example. Assume that the type analyzer is performing the syntactic analysis of a typed model schema. The following labels have been introduced in the type context and have been assigned ordinal positions within their categories:

@CONCEPTS	QUANTITIES	UNITS
1. @BUTTER	1. WEIGHT	1. LBS
2. @GUNS	2. DURATION	2. DAYS
3. @TIME	3. COST	3. US \$
4. @OBJECTIVE	4. VOLUME	4. FEET
5. CARDINALITY	5. BOXES	

Part of the type analyzer's responsibility during the Syntactic Analysis stage is to construct canonical types for parameters, variables and functions. Suppose it encounters a type statement for a parameter called "SCOST(l,j),"

**<< COST of @OBJECTIVE / (WEIGHT of @BUTTER / DURATION of@TIME)
US_\$ / ([100] LBS / DAY) # >>.**

and must store it in a canonical form.

The first step in creating a canonical form is to identify the different dimensional clauses within the type statement and the arithmetic operators which bind them together. In this case, there are three clauses: "COST of @OBJECTIVE", "WEIGHT of @BUTTER" and "DURATION of @TIME." The clauses are bound together by three division operators, "/". Next, the type analyzer rewrites this complex fraction of dimensional clauses as a simple product of clauses with exponents of "1" or "-1":

(COST of @OBJECTIVE)¹ * (DURATION of @TIME)¹* (WEIGHT of @BUTTER)⁻¹.

After this simplification occurs, the type analyzer creates a *concept vector* to store the concepts described in each clause. A concept vector is composed of "0"s and "1"s. The order of its elements correspond to the ordinal position imposed by the type analyzer on concept labels. The concept vector for SCOST(i,j) is

@BUTTER	@GUNS	@TIME	@OBJECTIVE
1	0	1	1

Notice that @GUNS is assigned a value of zero because there is no dimensional clause with that concept in this numerical type. If more than one dimensional clause within a type were to contain the same concept, it would still appear only once in the concept vector.

A *quantity vector* is a vector of signed integers used to represent the quantity contained in each dimensional clause of a numerical type. Each element of this vector corresponds to the exponent a particular label would be assigned if the quantity were expressed in product form. The order of the elements in a quantity vector is the order determined for quantity labels by the type analyzer.

To guarantee a unique representation, the signs of quantity vector elements are related to the sign of the exponent of their dimensional clause. If the dimensional clause has a negative exponent, the signs of its quality vector exponents are reversed. Thus, a dimensional clause like " $(A^1 \cdot B^{-1} \text{ of } @C)^{-1}$ " would be considered equivalent to " $(A^1 \cdot B^1 \text{ of } @C)^1$." The basis for this design decision is that concepts are existential: we have no notion of raising existential things to powers (*length of @HOUSE * length of @HOUSE is length² of @HOUSE, not length² of @HOUSE²*).

The quantity vectors for the dimensional clauses of "SCOST(i,j)" are the rows of the matrix given below.

	WEIGHT	VOLUME	COST	DURATION	CARDINALITY
@BUTTER	-1	0	0	0	0
@TIME	0	0	0	1	0
@OBJECTIVE	0	0	1	0	0

A one-to-one relationship exists between a non-zero element in a numerical type's concept vector and a quantity vector. If two or more dimensional clauses in a type were to contain the same concept, their quantity vectors would be summed together to form a single vector. In the event that a quantity vector becomes a zero vector, it is eliminated from the canonical form and its associated concept vector element is given the value "0." This is equivalent to manual "canceling" of terms.

The last step in creating a canonical type is to represent the unit component. This involves reduction of complex fractions of unit clauses into two simpler representations: a *units vector* and a *scale factor doublet*. A units vector is constructed in the same fashion as a quantity vector. Its entries are signed integer exponents, sequenced in the order assigned to unit labels by the type analyzer. The scale factor doublet is a pair of integers that represent a rational number. It is formed by computing the product of unit clause scale

factors (the reciprocal of a unit scale factor is used if the unit clause appears in the denominator of unit component fraction) expressed as a pair of signed integers.

Our decision to specify scale factors as rational numbers rather than decimal numbers (Figure 4.5) was made to facilitate the determination of equivalence by a computer. The dominant format for representing decimal numbers on a computer is called the floating point number system. The details of floating point implementation vary from machine to machine but the same basic principles apply in all cases. A floating point representation of a decimal number is analogous to scientific notation where a decimal number is written as a signed mantissa multiplied by a power of 10. In floating point notation, a decimal number is stored as a signed fraction multiplied by an integer power of a machine-dependent base, usually some power of 2. All numbers cannot be represented exactly and some error is incurred if an attempt is made to store such a number. For example, the decimal number .01 is not representable in a finite number of bits in bases 2, 8, 16 or 32.

Arithmetic with floating point decimal numbers is not commutative. When floating point decimal numbers are added, subtracted, multiplied or divided, the result may be a non-representable number. Non-representable numbers are approximated by a floating point surrogate by a pre-determined rule. Our decision to specify scale factors as rational numbers and to manipulate them as integer pairs preserves commutativity, enabling derived types to be compared without inaccuracies caused by loss of precision. Although we have specified quantities to have only integer exponents, a similar scheme could be used to allow quantity exponents to assume rational values. The units vector and scale factor doublet for "SCOST(i,j)" appear below.

Units Vector:	LBS	DAYS	US \$	FEET	BOXES
	-1	1	1	0	0

Scale factor Doublet:	NUMERATOR	DENOMINATOR
	1	100

To determine whether two types expressed in canonical form are equivalent, the type analyzer begins by comparing their concept vectors. If the concept vectors are not identical, no further checks are necessary. If the concept vectors match, corresponding quantity vectors are then compared for equality, followed by units vectors and scale factor doublets. Any difference indicates that the compared types are not equivalent.

Type addition and type subtraction are performed by determining that both operands are type equivalent and then assigning the common type to the resultant. The multiplication or division of a valid type by another valid type is always a valid type. Thus, type multiplication and type division, on the other hand, create new types dynamically.

We will describe how numerical types expressed in canonical form can be manipulated to implement type multiplication first. Figures 4.7 and 4.8 depict the canonical forms of SCOST(i,j) and a variable, "FLOW(i,j)," respectively. The numerical type statement of FLOW(i,j) is

<< WEIGHT of @BUTTER / DURATION of @TIME # [50] LBS / DAY >>.

For purposes of this example, assume that SCOST(i,j) is the multiplicand and FLOW(i,j) is its multiplier.

The first step in multiplying one type by another is to create a concept vector for their product. This is done by performing a logical "OR" operation between corresponding elements of the multiplier's concept vector and the multiplicand's concept vector. Next, a quantity vector is constructed for each non-zero entry in the product's concept vector. This is done by summing the quantity vectors associated with corresponding entries in the concept vectors of the multiplier and multiplicand. Once the dimensional component of the new type has been represented, attention is turned to its unit component. The units vector

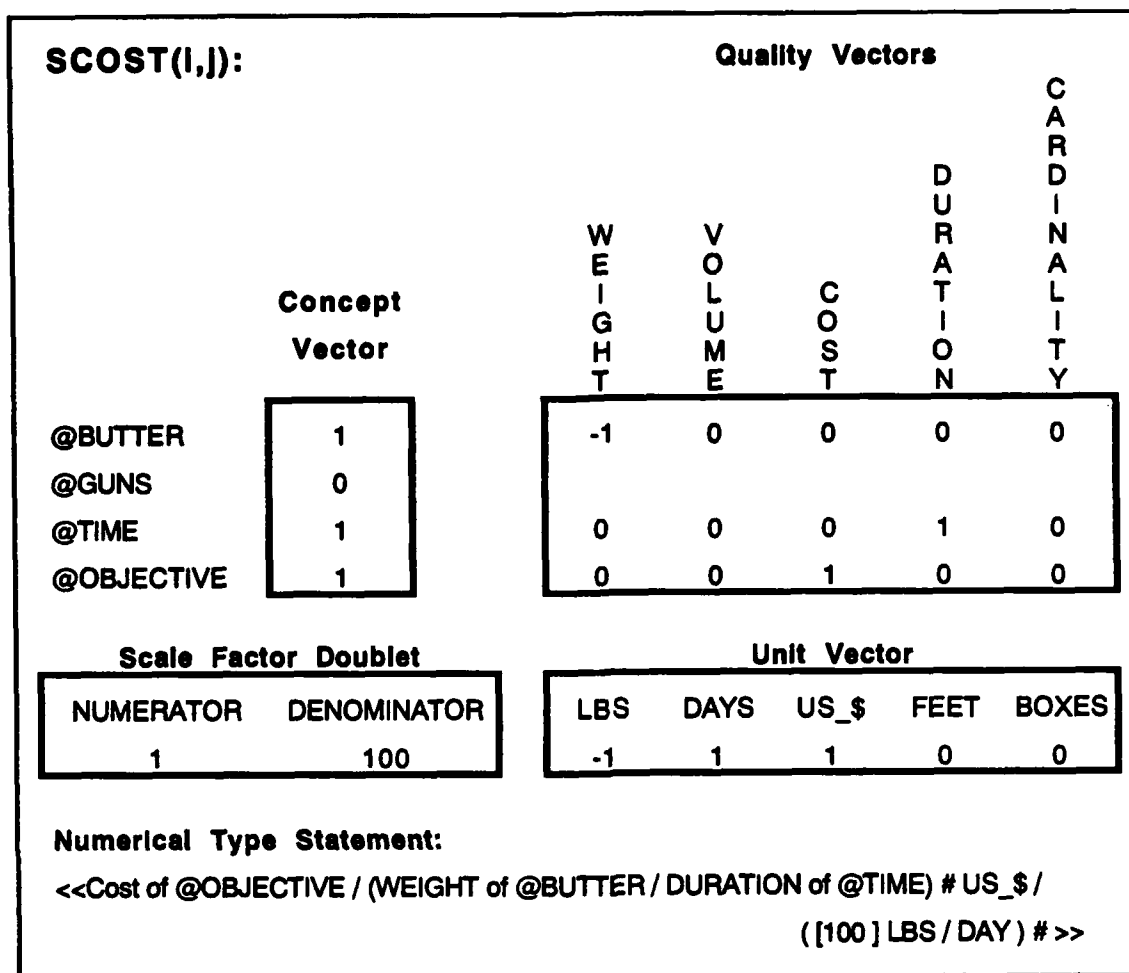


Figure 4.7 Canonical Form for SCOST(i,j) Quality Vector

of the product is obtained by summing the units vectors of the multiplier and multiplicand. Finally, the scale factor doublet is computed by multiplying the numerator element (denominator element) of the multiplicand by the numerator element (denominator element) of the multiplier. The new doublet is then normalized by dividing each element by their greatest common divisor. Figure 4.9 contains the canonical form for the product of SCOST(i,j) and FLOW(i,j), as well as its type language statement equivalent.

Division of one type by another is accomplished in a similar fashion. First, a concept vector for the quotient is created in the same manner as multiplication. Next, a

FLOW(I,J):		Quality Vectors				
	Concept Vector	W E I G H T	V O L U M E	C O S T	D U R A T I O N	C A R D I N A L I T Y
@BUTTER	1	1	0	0	0	0
@GUNS	0					
@TIME	1	0	0	0	-1	0
@OBJECTIVE	0					
Scale Factor Doublet		Unit Vector				
NUMERATOR	DENOMINATOR	LBS	DAYS	US_\$	FEET	BOXES
50	1	1	-1	0	0	0
Numerical Type Statement:						
<<(WEIGHT of @BUTTER / DURATION of @TIME) # ([50] LBS / DAY) # >>						

Figure 4.8 Canonical Form for FLOW(i,j) Quality Vector

quantity vector is created for each non-zero entry in the quotient's concept vector. Instead of summing corresponding quantity vectors as was done in type multiplication, the quantity vectors of the quotient are computed by subtracting the quantity vectors of the divisor from the quantity vectors of the dividend.

The last step in constructing a canonical form for a quotient of two types is to create a units vector and its scale factor doublet. The units vector of the quotient is constructed by subtracting the units vector of the divisor from the units vector of the dividend. The scale factor doublet is constructed by multiplying the numerator entry (denominator entry) of the dividend's doublet by the denominator entry (numerator entry)

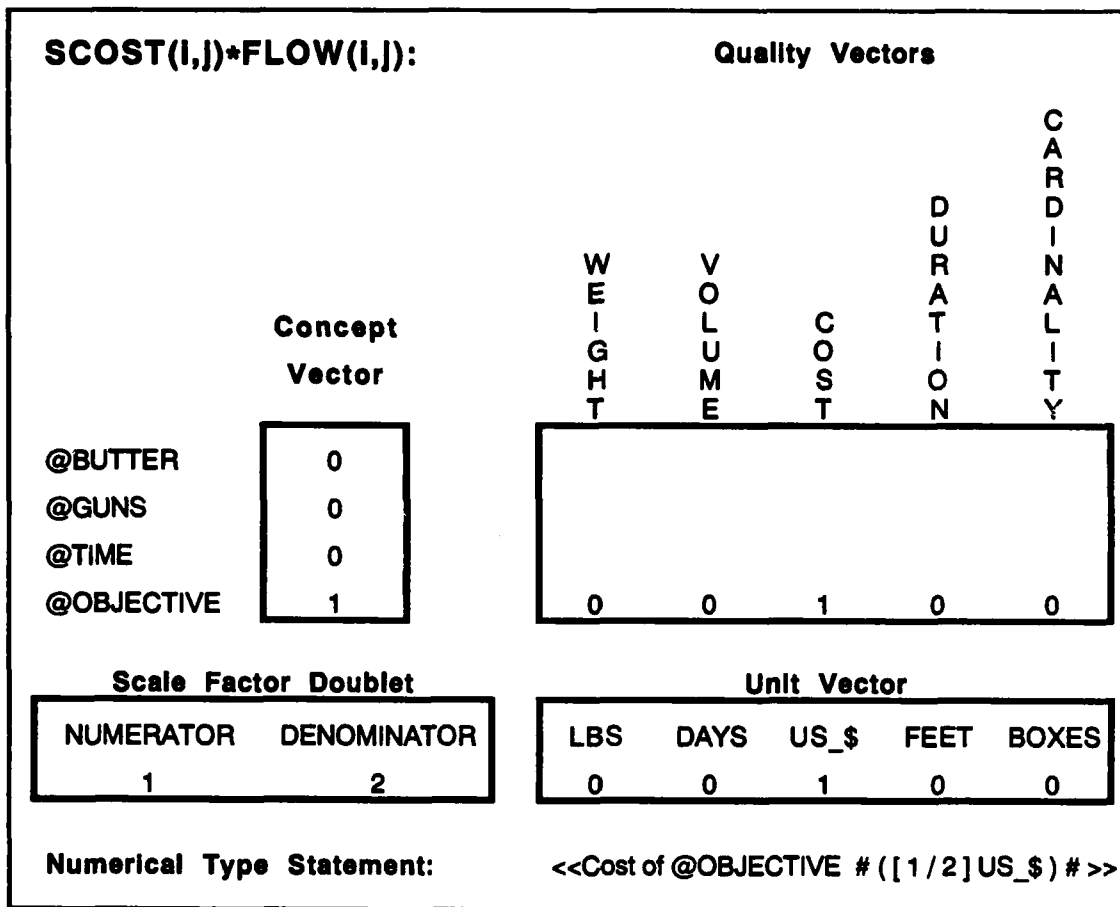


Figure 4.9 Canonical Form for SCOST(i,j)*FLOW(i,j)

of the divisor's doublet. Each element of the new doublet is then normalized by their greatest common divisor. If "SCOST(i,j)*FLOW(i,j)" is used as a dividend, and FLOW(i,j) is used as its divisor, the procedure just described reproduces the canonical form of SCOST(i,j) depicted in Figure 4.7.

3. Type Conversions

Types in canonical form can be judged to be different for one or more of the following reasons. They can contain different concepts; they attribute different quantities to the same concept; or, they differ in their units. These kinds of errors in an algebraic expression may be indicative of a fundamental flaw in the model. Either the offending

expression is inconsistent with the type declarations provided by the modeler or vice versa. Removal of this inconsistency may require that type declarations of variables and parameters be revised; it may even require that an entire constraint or set of constraints be reformulated.

In other cases, numerical type errors are the result of modeling omissions rather than explicit modeling errors. The modeler fails to include a multiplicative *conversion* for transforming either one type to the other or for transforming both types to a common type. These conversions are valid when they are based upon identities which map one measurement system into another, obey dimensional laws, or satisfy precepts in the modeled problem. We now offer several examples of how the EML and the type language could be used to implement type conversions.

Suppose a variable, "FLOW(i,j)," is compared to a parameter, "UP_BD(i,j)," and a numerical type error occurs. The declarations of FLOW(i,j) and UP_BD(i,j) are given below. The type statements of both objects have identical dimensional components, but they differ in their units. FLOW(i,j) is measured in "LBS / DAY" while UP_BD(i,j) is measured in "TONS / WEEK."

```
VARIABLES  FLOW(i,j) {ARC} ; << WEIGHT of @BUTTER / DURATION of
                                     @TIME # LBS / DAY # >>
```

```
PARAMETERS  UP_BD(i,j) {ARC} ; << WEIGHT of @BUTTER / DURATION
                                     of @TIME # TONS / WEEK # >>
```

```
CONSTRAINTS  CAP(i,j) {ARC} := FLOW(i,j) =L= UP_BD(i,j);
```

Since "LBS" is proportional to "TONS" and "DAY" is proportional to "WEEK", an EML parameter with an appropriate type statement can be declared and used in constraint "CAP(i,j)." The new parameter, "SCALE_DN," is defined below and used to transform the type of FLOW(i,j) to the type of UP_BD(i,j) in CAP(i,j). The choice of which object to convert is arbitrary since the relationship between LBS / DAY and TONS / WEEK is invertible.


```
PARAMETERS  SCALE_DN / 0.0035 /; << @* # (TONS/WEEK) /
                                                (LBS/DAY) # >>
```

```
CONSTRAINTS  CAP(I,J) {ARC} := SCALE_DN * FLOW(I,J) =L= UP_BD(I,J) ;
```

Notice that the type statement of SCALE_DN has a universal dimensional component. Thus, when the type of SCALE_DN and the type of FLOW(I,J) are multiplied, the product has the dimensional component of FLOW(I,J): "WEIGHT of @BUTTER / DURATION of @TIME". Notice also that the numerical constant needed to transform the numerical value of FLOW(I,J), "0.0035", is included in the EML declaration of SCALE_DN.

Multiplicative type conversions can also be written in the type language to resolve dimensional differences. Consider the following model excerpt:

PARAMETERS

```
BASE ; << BASE_AREA of @SOLID # METER^2 # >>
```

```
OVERHEAD ; << SIDE_HEIGHT of @SOLID # METER # >>
```

FUNCTIONS

```
CAPACITY := BASE*OVERHEAD ; << REC_VOLUME of @SOLID
                                # METER^3 # >>
```

In its present form, "CAPACITY" would generate a numerical type error because the quantities declared in its type statement are different from those derived from its algebraic definition. The intention of the modeler is clear: the volume of a solid is calculated by multiplying its base area by its height. The following parameter documents this identity and performs the required conversion. Notice that the unit component of this multiplier is "UNITY". Thus, no changes are made in the unit component of the algebraic definition. "VOL_IDENTITY" is assigned a numerical value of "1.0" so that the type transformation will not affect the numerical value of CAPACITY.

PARAMETERS

```
VOL_IDENTITY / 1.0 / ; << REC_VOLUME / BASE_AREA  
*SIDE_HEIGHT of @SOLID # UNITY # >>
```

Our next example shows that type conversions can also be used to reconcile concept differences. Below are EML and type language declarations of two variables and an out-of-context EML algebraic expression. We can tell from their type statements that summing "ROME(I)" and "NAVEL(I)" will result in a numerical type error because their quantities attribute different concepts. The intention of the modeler in formulating this expression is to suppress the inherent differences between apples and oranges and focus, instead, on their common attributes as fruit. This form of abstraction is called *generalization*.

VARIABLES

```
ROME(I) {GROCERS} ; << WEIGHT of @APPLES # LBS # >>
```

```
NAVEL(I) {GROCERS} ; << WEIGHT of @ORANGES # LBS # >>
```

We define two parameters which will convert these different types into a common new type. After these conversions are applied, the type of the sum will document the intention of the modeler.

PARAMETERS

```
APPLES_TO_FRUIT / 1.0 /; << WEIGHT of @FRUIT / WEIGHT of  
@APPLES # UNITY # >>
```

```
ORANGES_TO_FRUIT / 1.0 /; << WEIGHT of @FRUIT / WEIGHT of  
@ORANGES # UNITY # >>
```

In summary, we have shown through the last three examples that each of the causes of numerical type error can be reconciled by defining type conversions. Type conversions make a model easier for model users to understand by making concept,

quantity and unit of measurement identities used by the modeler an explicit part of the model.

4. Type Coercions

Each of the type transformations posed in the last section were valid because they were based upon relationships between systems of measurement, dimensional conventions or other identities. We now consider how this kind of information could be included explicitly in the model and used to transform types automatically. Conversions which are applied autonomously by the type analyzer are called *coercions*.

a. Units

Recall that a system of measurement is established by assigning units to the basic quantities of a dimensional system. For example, inches may be assigned to measure length. All other units within the measurement system are then derived according to dimensional rules. Since area can be defined as length^2 , the measure assigned to area would be inches^2 . Different systems of measurement based on the same dimensional system support the notion of *commensurability*. Two things are defined to be commensurable if one is a constant multiple of the other. Inches, feet, centimeters, and meters are all commensurable with one another. A numerical type error due to a difference in units can always be corrected if the units involved have this property.

Determining whether two units or two unit expressions are commensurable is equivalent to asking whether their ratio is commensurable with unity. For example, if a conversion from LBS/DAY to TONS/WEEK were required, we would ask whether " $(\text{TONS/WEEK}) / (\text{LBS/DAY})$ " is commensurable with unity. By the identities " $1 \text{ WEEK} = 7 \text{ DAYS}$ " and " $2000 \text{ LBS} = 1 \text{ TON}$ ", this ratio has the value 0.0035. Hence, the two units are commensurable and the multiplier for converting the numerical value associated with LBS/DAY to TONS/WEEK is 0.0035.

It is well known, for example Karr and Loveman <1978>, that given a finite set of units, a non-redundant, finite set of identities can be written that capture all commensurable relationships. In addition, the commensurability of arbitrary products of these units raised to rational powers can be determined by linear algebra.

To illustrate this, we define the following set of units:

(HOUR, DAY, WEEK, LBS, TON, FOOT, ACRE).

The identities

$$\text{DAY} = 24 \text{ HOUR}$$

$$\text{WEEK} = 7 \text{ DAY}$$

$$\text{TON} = 2000 \text{ LBS}$$

$$\text{ACRE} = 43560 \text{ FOOT}^2$$

are sufficient to describe all the commensurable relationships within this set.

Next, we create a coefficient matrix for manipulating these identities. The essential point in this procedure is to treat a unit like a multiplying variable that obeys associativity, commutativity and this rule:

When two identical variables are multiplied together, exponents are added.

We begin by taking the symbolic logarithm of each identity:

$$\text{LN (DAY)} = \text{LN (24)} + \text{LN (HOUR)}$$

$$\text{LN (WEEK)} = \text{LN (7)} + \text{LN (DAY)}$$

$$\text{LN (TON)} = \text{LN (2000)} + \text{LN (LBS)}$$

$$\text{LN (ACRE)} = \text{LN (43560)} + 2 \text{ LN (FOOT)} .$$

Next, we rewrite each equation in homogeneous form, placing terms without units at the far right.

$$\text{LN (DAY)} - \text{LN (HOUR)} - \text{LN (24)} = 0$$

$$\text{LN (WEEK)} - \text{LN (DAY)} - \text{LN (7)} = 0$$

$$\begin{aligned}\text{LN (TON)} &- \text{LN (LBS)} - \text{LN (2000)} = 0 \\ \text{LN (ACRE)} &- 2 \text{ LN (FOOT)} - \text{LN (43560)} = 0\end{aligned}$$

The third step is to store these transformed identities in matrix form. Each identity is a matrix row. The coefficients of each symbolic logarithm unit in an identity are written in the given order of the unit set, in this case: HRS, DAY, WEEK, LBS, TON, FOOT, ACRE. We will refer to the sub-matrix formed from these labeled columns as "P." The last entry in each matrix row is the natural logarithm of a real number. We will refer to this column as "v." The completed P|v matrix for this example appears below.

HRS	DAY	WEEK	LBS	TON	FOOT	ACRE	
-1	1	0	0	0	0	0	-LN (24)
0	-1	1	0	0	0	0	-LN (7)
0	0	0	-1	1	0	0	-LN (2000)
0	0	0	0	0	-2	1	-LN (43560)

The hypothesis that one unit is commensurable with another can be expressed as a vector in a similar fashion. The next three lines show the transformation of the statement "LBS/DAY is commensurable with TONS/WEEK" from algebraic form to vector form. We will refer to (3) below as "c." Notice that c has the same dimension as a row of P.

- (1) $\text{LBS} \cdot \text{DAY}^{-1} = \text{TONS} \cdot \text{WEEK}^{-1}$
- (2) $\text{LN (LBS)} - \text{LN (DAY)} - \text{LN (TONS)} + \text{LN (WEEK)} = 0$
- (3)

HRS	DAY	WEEK	LBS	TON	FOOT	ACRE
0	-1	1	1	-1	0	0

The procedure for determining whether two units are commensurable is stated in Figure 4.10 in terms of linear algebra. Demonstrating that c is a member of the vector space spanned by the rows of P is equivalent to showing that all unit exponents cancel

Let H be the hypothesis that the ratio of unit A to unit B is commensurable with 1;

P be an m by n matrix (derived from unit identities as described above) where n and m are the number of unit labels and unit identities declared in the type context of a model, respectively;

v be an m by 1 vector;

w be a 1 by m vector of unknowns; and

c be an n by 1 vector derived to test the hypothesis.

H is true if there exists a solution, w , to the system of equations

$$w P = c.$$

If w exists, the requisite multiplier, k , for converting unit A to unit B is given by

$$k = e^{w \cdot v}.$$

Figure 4.10 Procedure For Determining Commensurate Relationships

when one unit is divided by another. This is the basis for an algorithm to determine if two units are commensurable and, if so, the value of the requisite multiplier.

A type language syntax for unit identities is given in Figure 4.11. These statements would appear after quantity statements in the type context to conform to the "define before use" principle.

b. Quantities

The methods we have applied to unit identities can be applied to quantity identities with minor modification. For example, seven identities were used in the engineering dimensional system described in Chapter 2:

- (1) mass = force * time² / length;
- (2) area = length²;
- (3) volume = length³;
- (4) velocity = length / time;

- (5) acceleration = length / time²;
- (6) pressure = force / length²; and,
- (7) energy = force * length.

The "P" or *postulate* matrix for these dimensional laws appears in Table 4.7. All commensurable relationships involving products of force, length, time, or any of the dimensions derived from force, length and time can be determined from this matrix.

The question of quantity commensurability, translated into linear algebra terms, is whether a vector of exponents extracted from the ratio of compared quantities is a member of the vector space spanned by the rows of the postulate matrix. Since dimensional laws have no pure number multipliers, no numerical conversion factor is computed.

The entries which cannot be eliminated in *c* could be used in an error message to describe the required relationship. For example, if a residual vector like the one shown below were produced, it would indicate that an identity involving density, weight and volume was required,

DENSITY	LENGTH	TIME	VOLUME	WEIGHT
1	0	0	1	-1

i.e., density * volume = weight. However, the indicated relationship could also involve from 2 to *n* quantities and/or be a violation of other known dimensional identities.

A type language syntax for quantity identities is given in Figure 4.11. These statements would appear after quantity statements in the type context to conform to the "define before use" principle.

c. Concepts

Since concepts are existential, concept identities are fundamentally different from the identities of quantities or units. Quantity and unit identities define synonyms,

<unit identity declaration> ::= "UNIT IDENTITY"
 (<unit label> "=" [<decimal number>|<scientific number>""]
 <unit label>["^"["-"]<unsigned integer>]
 ("*"<unit label>["^"["-"]<unsigned integer>]*";")+

Example: UNIT IDENTITY
 FOOT = 12.0 INCH;
 KILOGRAM = NEWTON*SEC^2*METER^-1;

<quantity identity declaration> ::= "QUANTITY IDENTITY"
 (<quantity label> "=" <quantity label>["^"["-"]<unsigned integer>]
 ("*"<quantity label>["^"["-"]<unsigned integer>]*";")+

Example: QUANTITY IDENTITY
 AREA = LENGTH^2;
 DENSITY = WEIGHT*VOLUME^-1;

<concept identity declaration> ::= "CONCEPT IDENTITY"
 (<concept label> "<--" "("<concept label>(","<concept label>)*"";")+

Example: CONCEPT IDENTITY
 @PHYSICAL_OBJECT <-- (@BUILDING,@CAR);
 @DAIRY_PRODUCT <-- (@BUTTER);

Figure 4.11 Identity Declarations

TABLE 4.7 POSTULATE MATRIX FOR DIMENSIONAL LAWS

	MASS	AREA	VOLUME	VELOCITY	ACCEL- ERA- TION	PRES- SURE	ENERGY	FORCE	LENGTH	TIME
(1)	-1	0	0	0	0	0	0	1	-1	2
(2)	0	-1	0	0	0	0	0	0	2	0
(3)	0	0	-1	0	0	0	0	0	3	0
(4)	0	0	0	-1	0	0	0	0	1	-1
(5)	0	0	0	0	-1	0	0	0	1	-2
(6)	0	0	0	0	0	-1	0	1	-2	0
(7)	0	0	0	0	0	0	-1	1	1	0

such as "velocity = length/time"; concept identities define generalizations, such as "@APPLES are a kind of @FRUIT." While synonyms are mutually replaceable, generalizations are not: apples may be fruit, but fruit are not necessarily apples. Another difference between quantity or unit identities and concept identities is the use of arithmetic operators. A new unit or a new quantity can be defined by applying arithmetic operators to existing quantities or to existing units. We have no notion of composing concepts, in an analogous way, to create new concepts.

Concept identities are written using an arrow operator, "<--", which can be read as "is a generalization of". For example, "@A <-- @B" is read as "@A is a generalization of @B." When multiple concepts have the same generalization, this can be expressed by separating them by commas and enclosing them in parentheses. For example, @A <-- (@B, @C, @D). In accordance with the "define before use" principle, a concept label must appear in a concept declaration or as a left-hand operand in a concept identity before it can be used as a right-hand operand in a concept identity. A BNF description of concept identity syntax is given in Figure 4.11.

We now describe how concept identities can be organized and manipulated to determine a concept coercion. Let each concept declared in the type context be a vertex in a graph, **G**. Let each concept identity define a directed edge, from a concept's vertex to the vertex of its generalization. Concept identities which group multiple concepts with a common generalizing concept define multiple edges. The graph corresponding to the concept identities given below is Figure 4.12.

CONCEPT IDENTITY	@F <-- @E ;
	@C <-- @B ;
	@E <-- (@B, @D) ;
	@B <-- @A ;
	@D <-- @A ;
	@I <-- @H ;

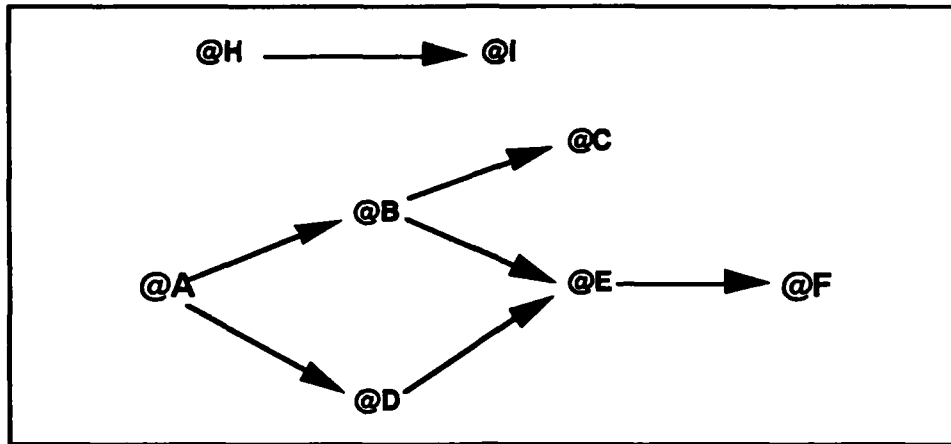


Figure 4.12 Concept Identity Graph

Suppose two types, T_1 and T_2 , are compared to determine the legality of an arithmetic operation and it is necessary to know whether a concept in T_1 can be reconciled with a concept in T_2 . All allowable coercions can be determined from the paths in G . For example, @A generalizes to @C because there is a directed path from @A to @C in G . @B and @D have common generalization @E because @E lies on paths from both @B and @D. We require G to be acyclic in order to avoid circular reasoning. Given this assumption, the complexity of doing this kind of testing is $O |E|$, where E is the set of edges in G (e.g., Aho, Hopcroft and Ullman <1974>).

Ambiguous situations can occur when two concepts have more than one common generalization. For example, @E and @F are both generalizations for @B and @D. Which one should be chosen as a concept coercion? We suggest two possible solutions. One alternative is to let the type analyzer select one according to some fixed rule, such as first encountered generalization. The other alternative is to report the possible choices to the modeler in the form of an error message and let him remove the ambiguity by redefining his identities.

d. Application of Numerical Coercions

The numerical coercion factors needed to make an arithmetic expression well-formed are determined by the type analyzer during type verification. They are applied by the modeling language translator during a subsequent step. The interface between the two systems is a revised form of the arithmetic expression that includes type analyzer-derived constants. In this section we describe how this revision could be created in an efficient computational form.

The order in which an arithmetic expression is evaluated depends upon the priority of its operators, the direction of evaluation, and the use of bracketing parentheses. A commonly used convention in arithmetic is to assign "*" and "/" equal but higher priority than "-" and "+"; and to evaluate from left to right. The order of evaluation can be altered by enclosing sub-expressions in parentheses. Figure 4.13 contains an unparenthesized and a parenthesized version of the same sequence of variables and operators, written in infix notation. By infix notation, we mean that sub-expressions have the form

<operand> <operator> <operand>

The numbers associated with the sub-expressions in Figure 4.13 indicate the order of evaluation. Notice that evaluating an infix expression requires repeated scanning in a left-to-right-manner.

Although modeling languages allow modelers to write arithmetic expressions in infix notation, they are not evaluated in this form. To avoid repeated scanning, each infix expression is first converted to an equivalent postfix or prefix expression and then evaluated. Prefix expressions are composed of sub-expressions which have the form

<operator> <operand> <operand>;

the sub-expressions of a postfix expression have the form

<operand> <operand> <operator>.

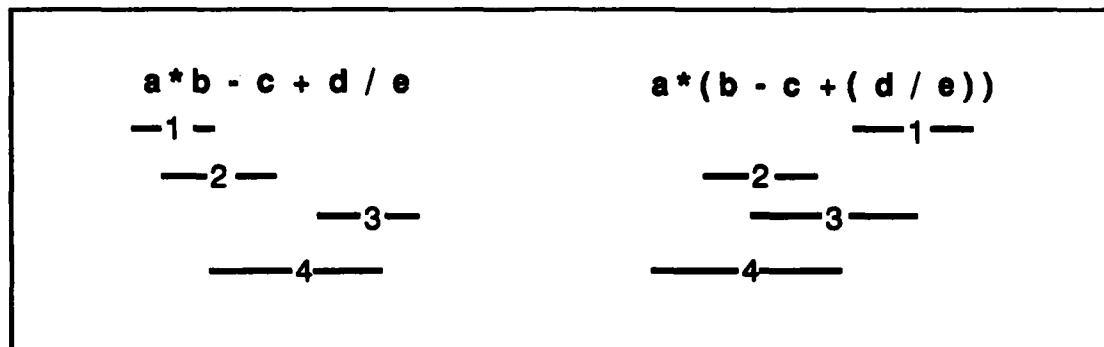


Figure 4.13 Examples of Infix Expression Evaluation

Algorithms for translating from infix to postfix or from infix to prefix are well-understood in the theory and practice of compiler writing (e.g., Tremblay and Sorensen <1985>). They have a complexity of $O(N)$ where N is the number of variable, operator and parenthesis tokens in the infix expression. In the remainder of this discussion we will concern ourselves with postfix notation. An analogous discussion holds for prefix notation. Examples of expressions in their infix, prefix and postfix form are given in Table 4.8.

Postfix notation has certain virtues that simplify the evaluation of expressions. First, postfix expressions are parenthesis free. Second, the priority of an operator is no longer relevant. The expression may be evaluated by making a single left-to-right scan, stacking operands, evaluating operators by removing the required number of operands from the stack, and placing each result onto the stack. An algorithm for postfix expression evaluation is given in Figure 4.14. A numerical example appears in Table 4.9.

The algorithm for postfix expression evaluation can be used for several purposes. When used for numerical evaluation, operand tokens are replaced by their numerical values and the rules of real number arithmetic are applied in the "APPLY_OPERATOR" procedure. The intermediate results pushed on the stack are numerical values. When used for type verification, the APPLY_OPERATOR procedure

TABLE 4.8 EQUIVALENT EXPRESSIONS

Infix	Prefix	Postfix
a	a	a
a+b	+ab	ab+
a+b-c	-+abc	ab+c-
a+(b-c)	+a-bc	abc-+
a/b/c	//abc	ab/c/
a*b-c	-*abc	ab*c-
a*(b-c)	*a-bc	abc-*
a+b*c-d	+a-*bcd	abc*+d-
(a+b)*(c-d)	*+ab-cd	ab+cd-*

manipulates canonical types according to the rules of type arithmetic. In this case, intermediate results pushed on the stack are the canonical types of sub-expressions. The algorithm can also be used to reassemble the original postfix input by concatenating operand and operator tokens together in postfix order each time the APPLY_OPERATOR procedure is called.

If a type analyzer and a modeling language translator implement postfix form in the same way, the APPLY_OPERATOR procedure used by the former can be modified to produce the revised postfix form needed by the latter. This is done as follows. Each time the APPLY_OPERATOR procedure is called, two steps are performed. First, the canonical types of the operands are manipulated according to the rules of type arithmetic. If a numerical coercion is required (to change scale and/or change units of measurement), the value of that multiplier is determined and assigned to a type analyzer-defined token. Second, the postfix form of the sub-expression is reconstructed. If no numerical coercion is required, the postfix form of the sub-expression is reassembled by concatenating operand and operator tokens in their original postfix order. If a numerical coercion is required, a revised postfix sub-expression is created in three steps. First, depending upon the conversion protocol, one of the two operands tokens is concatenated to the token

Algorithm EVAL_POSTFIX.

This algorithm computes the value *RSLT* of an input string *POSTFIX* which contains a postfix expression. It is assumed that the last character in *POSTFIX* is a delimiter, ";", and that *POSTFIX* contains no more than *n* tokens.

A procedure **NEXT_TOKEN(POSTFIX,n)** is used to extract the next *TOKEN* from *POSTFIX*. A token is either an operand, an operator or a delimiter.

STACK(1:n) is a one dimensional array used as a stack.

PUSH(TOKEN,STACK,n,TOP) is a procedure that places *TOKEN* on the top of the stack and updates the index, *TOP*, which points to the top element of the stack.

POP(STACK,n,TOP) is a function that removes the top element of the stack and returns its value.

APPLY_OPERATOR(x,STACK) is a procedure used to remove the correct number of operands for operator *x* from the stack, perform the required operation, and store the result on the stack.

```
BEGIN
  TOP <- 0 // Initialize STACK //
  LOOP:
    TOKEN <- NEXT_TOKEN( POSTFIX )
    CASE
      : TOKEN = ";" : RSLT <- POP(STACK,n,TOP)
        RETURN // evaluation complete //

      : TOKEN = operand : PUSH(TOKEN,STACK,n,TOP)

      : ELSE: // x = operator //
        APPLY_OPERATOR(TOKEN,STACK)
    END
  END EVAL_POSTFIX
```

Figure 4.14 Algorithm for Evaluating Postfix Expressions

assigned to the coercion and a multiplication token, say "*", in postfix order. For example, suppose the two operands are sub-expressions "a b + c d * -" and "x y /" in the constraint "a b + c d * - x y / =E=" and a numerical coercion "K" has been calculated to convert the units of the left operand to those of the right operand to make the "=E="

TABLE 4.9 Example of Postfix Expression Evaluation

Evaluation of 3 4 + 5 6 + * ; <==> (3 + 4) * (5 + 6);		
Current Input Symbol	Contents of Stack Rightmost Symbol is Top of Stack	Action
3		
4	3	
+	3 4	APPLY + to 3 4
5	7	
6	7 5	
+	7 5 6	APPLY + to 5 6
*	7 11	APPLY * to 7 11
;	77	POP STACK

operation well-formed. The revised form of the left operand would be written in postfix order as

" a b + c d * - K * ".

Next, the modified operand is placed in the same relative position in the original postfix expression and concatenated to the other operand. Finally, the operator token is added as the right-most token in the expression. In this case, the revised postfix expression is

" a b + c d * - K * x y / =E= ".

The reconstructed postfix expression and its canonical type are then placed on the stack for use in a subsequent operation. When the delimiter is encountered in the input, the canonical type and the reassembled postfix expression are on the top of the stack. If the type is not a numerical type error, the type analyzer forwards the reassembled (possibly revised) postfix expression to the modeling language translator to be generated. If the expression is stored as a tree, an analogous analysis can be done (see Natchsheim <1987>).

C. INDEX TYPE SYNTAX AND SEMANTICS

1. Fundamentals

Index typing formalizes the ordinal and arithmetic properties assigned to index sets by modelers. The type of an index set determines the operators that can be legally applied to its elements and the properties of sets which include its elements as constituent parts. The lexical and syntactic conventions of index type declaration are simple. An *index type statement* (Figure 4.15) consists of a single word (nominal, ordinal, or ordinal+). It follows the declaration of each EML index set and is a required part of a typed schema. Operations defined on indices and on index sets in EML are imperatives used to evaluate index types.

`<index type statement> ::= "<<" "nominal" | "ordinal" | "ordinal+" ">>"`

Figure 4.15 Index Type Declaration

A calculus for index types depends upon the definition of an *order relation* between index set elements and on the function defined to map elements of ordered index sets to the positive integers. The order relation and index set element-to-integer function for our type language is defined in Figure 4.16. The elements of an index set declared as ordinal or ordinal+ are ordered in the sequence in which the individual elements first appear in the schema. Figure 4.17 lists definitions of nominal, ordinal and ordinal+ index types.

2. Index Operators

Table 4.10 list the operators used in EML to form index expressions and index relations (See Appendix A). These constructs are used to identify elements, to convert the ordinal position of an element to an integer for arithmetic computation and to form

predicates to control the EML "SUM" function and "SELECT" function⁵. Table 4.11 is a notational key for the element operator semantic rules presented in Table 4.12.

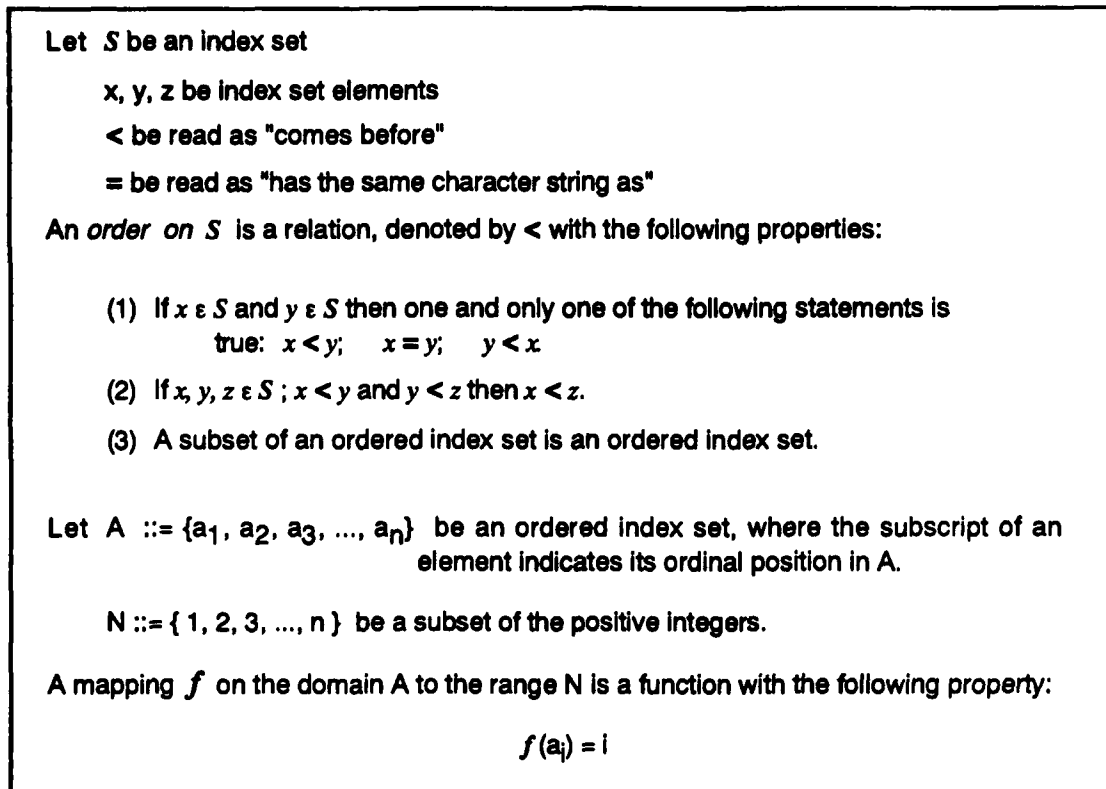


Figure 4.16 Order Relation and Element-to-Integer Function Definitions

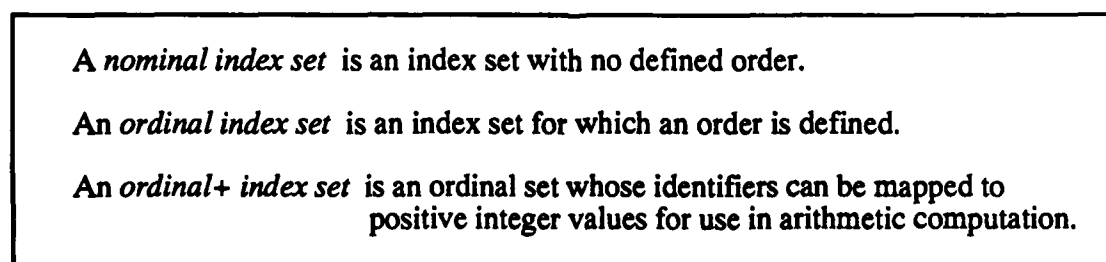


Figure 4.17 Index Type Definitions

⁵The "SUM" function is used in MLs as the " Σ " symbol is used in conventional mathematical notation. The "SELECT" function is an operator used to extract tuples that satisfy specified conditions from an index set.

TABLE 4.10 INDEX OPERATORS

Name	Description	Precedence
++	successor	1
--	predecessor	1
POSITION()	ordinal position	2
EQ	equal to	3
NE	not equal to	3
LT	less than	3
LE	less than or equal to	3
GT	greater than	3
GE	greater than or equal to	3

The symbols listed in Tables 4.11 and 4.12 are primitives that are used to define the syntax and semantics of our index type language

TABLE 4.11 NOTATION KEY

Interpretation	Symbol
Universal Concept	@*
Quantities	UNITY, BOOLEAN
Symbolic Index Operands	I_1, I_2
Symbolic Index Set Operands	S_1, S_2
Unsigned Integer	N
Symbolic Index Expression Resultant	IR

TABLE 4.12 INDEX OPERATOR TYPE LANGUAGE SEMANTICS

Operator	Production	Semantic Rule
++, --	IR <- I_1 ++ N IR <- I_1 -- N	IR.type ::= IF I_1 .type = ordinal THEN ordinal ELSE IF I_1 .type = ordinal+ THEN ordinal+ ELSE index type error
POSITION ()	IR <- POSITION(I_1)	IR.type ::= IF I_1 .type = ordinal+ THEN @* # UNITY # ELSE index type error
EQ, NE	IR <- I_1 EQ I_2 IR <- I_1 NE I_2	IR.type ::= @* # BOOLEAN #
LT LE GT GE	IR <- I_1 LT I_2 IR <- I_1 LE I_2 IR <- I_1 GT I_2 IR <- I_1 GE I_2	IR.type ::= IF (I_1 .type = ordinal OR I_1 .type = ordinal+) AND (I_2 .type = ordinal OR I_2 .type = ordinal+) THEN @* # BOOLEAN # ELSE index type error

3. Set Operators

Table 4.13 lists the five operators used in EML to construct sets of n -tuples from simple index sets. The "UNION", "DIFFERENCE" and "CROSS" operators provide the traditional set operations of union, difference and Cartesian product. The "SELECTION" and "PROJECTION" operators are special operators used to make queries of databases organized according to the relational data model (see Ullman <1982>). Given a collection of domains $D_1, D_2, D_3, \dots, D_n$, a *relation* is a set of ordered n -tuples $(d_1, d_2, d_3, \dots, d_n)$ where $d_1 \in D_1, d_2 \in D_2, d_3 \in D_3, \dots, d_n \in D_n$. A domain is simply a set of values.

In modeling languages, a subscripted numerical object (parameter, variable, function, or equation) can be thought of as a relation. Each instance is uniquely identified by a tuple of elements drawn from each referenced index set. For example, one instance of the parameter set defined as

PARAMETERS
SCOST(I,J) {PATHS}; << COST of @OBJECTIVE / (WEIGHT of @BUTTER
/ DURATION of @TIME) # US_\$ / ([100] LBS / DAY) # >>

could be the tuple (New York, San Francisco). Designers of modeling languages (i.e., Bisschop and Meeraus <1982>, Geoffrion <1988>) have recognized the similarity between the use of relations by database designers to abstract information in a database and the use

TABLE 4.13 SET OPERATORS

Name and Syntax	Description
UNION(S_1, S_2)	set union
DIFFERENCE(S_1, S_2)	set difference
CROSS(S_1, S_2)	Cartesian product
PROJECTION(S_1 [<index tuple>])	relational algebra projection
SELECTION(S_1 [<index relation list>])	relational algebra selection

of indexed components by modelers to abstract the structure of models. Consequently, relational algebraic operators have been adopted to varying degree in modeling languages to manipulate indexed model objects.

"SELECTION" is an operator used to extract a subset of n -tuples from within a relation that satisfy a predicate. In EML, the predicate is an <index relation list> (see Appendix A). The "PROJECTION" operator is used to construct a subset of k -tuples, from a relation of n -tuples, $k < n$. In EML, the indices to be retained in a projection operation on an n -dimensional index set are specified as an <index tuple>. (See Appendix A for a BNF description of <index relation list> and <index tuple>).

When designing a type language for index sets, we are faced with a choice among alternative rules for combining ordered index sets. One rule would be to order the elements lexicographically. Another rule would be to concatenate the elements of one set to the other according to a syntactic convention. For example, if **A** and **B** are ordered index sets based upon different order relations, the ordered index set resulting from the modeling language statement "**UNION(A, B)**" would list the elements of **A** first in sequence since **A** was the first operand encountered in the statement. The rules for combining index sets adopted for our type language are listed below.

- (1) When sets derived from the same ordering relation are combined by index set operators, the elements of the resulting set are ordered according to the ordering relation shared by the parent index sets.
- (2) When sets derived from different ordering relations are combined by index set operators, the elements of the resulting set are unordered.

V. LANGUAGE EXTENSIONS

A. POLYMORPHIC TYPES

The purpose of a type system in a programming language is to constrain the way objects may interact with other objects. To this end, we have imposed a type structure on indices and on numerically-valued symbols (constants, parameters, functions, variables and constraints). Once type statements have been made for these primitives by the modeler, the type analyzer is able to deduce the types of arithmetic expressions within functions and constraints and to determine the legality of index suffixes and iterated arithmetic operations.

Programming languages in which the type of every expression can be determined before imperative statements are executed are said to be *statically typed*. Static typing facilitates early detection of typing errors and makes executing programs more efficient by eliminating the need for run-time checks. In addition, it enforces programmer discipline that makes programs easier to read.

The requirement that all program variables and expressions be bound to a type at compile-time is sometimes too restrictive. In some programming languages, such as ML (Milner <1984> as referenced in Cardelli and Wegner <1986>), it is replaced by a weaker requirement that all program expressions be guaranteed to be type consistent although the type itself may be statically unknown. This feature enables generic procedures to be written, such as a sorting program that works on any type with an ordering relation.

In this section we propose a weaker equivalence criterion for numerical types. This relaxation increases the flexibility and expressive power of a type language by allowing parameters, variables, functions and constraints to be associated with more than one type. However, these benefits are not accrued without cost. A weaker equivalence criterion

limits the kinds of checks that can be made before the types of these objects are completely specified.

Certain classes of mathematical programming models, such as pure networks, can be considered to be generic. Their identity depends upon the relationships that the variables and technical coefficients have to one another. The real-world interpretation of these objects is superfluous to the dimensional consistency of the constraints and objective function as long as these dependencies are satisfied. Consider the linear programming constraint

$$\text{SUM } (j) \{j\} (A(i,j) * X(j)) = L = B(i).$$

As long as the relationship

$$A(i,j) \text{ equals } 1 \text{ } B(i) \text{ unit} / 1 \text{ } X(j) \text{ unit}$$

holds, we know by inspection that the constraint compares like things.

We propose to accommodate this sort of relationship within the type language by allowing a modeler to declare a named numerical object to have a *polymorphic type*. By "polymorphic," we mean that the object may assume one of the completely specified, or *monomorphic*, types which can be constructed from the concepts, quantities and units of measurement declared in the type context. The monomorphic type used to fix the dimensional and unit characteristics of a data object would be made available to the type analyzer after type evaluation but before data entry. Figure 5.1 provides examples of the polymorphic type declaration of a linear programming constraint and objective function. Notice that the cost coefficient, "C(j)", has a type statement which is a mixture of the grammar we defined earlier and a polymorphic type. In general, the two kinds of numerical types can be convolved using multiplication and division. Polymorphic types can be compounded by multiplication, division and integer exponentiation.

VARIABLE	$X(j) \{j\} ; \ll \text{TYPE } X(j) \gg$
PARAMETER	$B(i) \{i\} ; \ll \text{TYPE } B(i) \gg$ $C(j) \{j\} ; \ll \text{COST of @OBJECTIVE \# US_\$ \# / TYPE } X(j) \gg$ $A(i,j) \{i \times j\} ; \ll \text{TYPE } B(i)/\text{TYPE } X(j) \gg$
FUNCTION	$\text{OBJ\$} := \text{SUM } (j) \{j\} (C(j)*X(j)) ; \ll \text{COST of @OBJECTIVE}$ $\text{\# US_\$ \#} \gg$
CONSTRAINT	$\text{TEST}(i) \{i\} := \text{SUM}(j) \{ \text{CROSS}(i,j) \} (A(i,j)*X(j)) = L = B(i) ;$ $\ll \text{TYPE } B(i) \gg$

Figure 5.1 Example of Polymorphic Typing

The dimensional consistency of expressions composed of objects with polymorphic types and/or objects with mixed numerical type declarations can be determined as follows. All monomorphic type components and polymorphic type components are subject to the rules of the type grammar for type arithmetic and type comparison operations. The type equivalence criterion for polymorphic types, however, is more stringent than that of monomorphic types. Whereas differences between expressions involving monomorphic types may be resolvable through identities declared within the type context, differences between polymorphic types are irreconcilable. Polymorphic types are subject to a *name equivalence* criterion instead of the structural equivalence criterion applied to monomorphic types. Under name equivalence, two objects do not have the same type unless they have the same name. For example, $\ll \text{TYPE } X(j) \gg$ is only equivalent to $\ll \text{TYPE } X(j) \gg$.

Although this amalgam of type checking protocols assures that all expressions accepted as well-formed are well-formed, it cannot make an equivalent guarantee about the expressions it rejects. For example, two EML parameters "X" and "Y" could be initially declared as having polymorphic types of $\ll \text{TYPE } X \gg$ and $\ll \text{TYPE } Y \gg$, respectively. At type evaluation the algebraic expression "X + Y" would generate an error because the name

In this section we consider how a taxonomy of concepts could be used to order quantities and reason about numerical types.

A taxonomy is a system for classifying a collection of things by generalization. At the lowest level, instances are grouped into classes in which some uniform conditions hold. Suppose we had a collection of animals. If the collection contained several instances of "big hopping things native to Australia," we could group them in a class called "kangaroo." When classes have properties in common, super-classes are formed. The classes "bandicoot," "wombat" and "kangaroo" would be instances of the class "marsupial" since all three kinds of animals nurture their unborn young in external pouches. If these classes are ordered from the most inclusive to the least inclusive and the resulting structure is a tree, the result is called a taxonomic hierarchy. More complex organizations result if a class can have more than one super-class. For example, "kangaroo" could be a sub-class of both "marsupial" and the class "animals that live in zoos."

Nearly all knowledge representation languages in artificial intelligence and object-oriented programming languages include some sort of taxonomic mechanism (e.g., SMALLTALK-80 [Goldberg and Robson <1983>]). There are two reasons for this popularity. First, it allows the programmer to describe each class as a specialization of a more generic class. This reduces the need to specify redundant information. It also simplifies maintenance since information need be entered and modified in only one place. The second advantage of taxonomic description is its conceptual parsimony. It enables a large collection of instances to be described in terms of a smaller collection of ordered classes.

Figure 5.3 is an example of a taxonomic hierarchy of concepts. The diagram has two organizational axes. The first axis associates quantities to individual concepts. These assignments further distinguish a concept from its siblings. The second organizational axis

relates each concept to its parent. Under an inheritance protocol, each concept inherits the quantities assigned to its generalizations. For example, "@AUTOMOBILE" inherits the quantities "HEIGHT", "LENGTH" and "WEIGHT" from "@PHYSICAL_OBJECT" and "FUEL_CAP" and "PASSENGERS" from "@MOTOR_VEHICLE". Thus, the dimensional description "LENGTH of @AUTOMOBILE" would be a legitimate construct in the type context described by this graph. The root node of the tree is the universal dimensional description, "@*." "@*" is always the most general concept in a type context, and is the "root node" in Figure 5.3.

We now show how relationships among concepts in this structure, or *concept graph*, could be used to resolve differences between types. Suppose "X" and "Y" are model variables with the dimensional components "WEIGHT of @AUTOMOBILE" and "WEIGHT of @CARGO", respectively. From a taxonomic viewpoint, the difference between these two descriptions is irreconcilable unless WEIGHT can be shown to be an assigned or inherited quantity of some common subsuming concept of @AUTOMOBILE

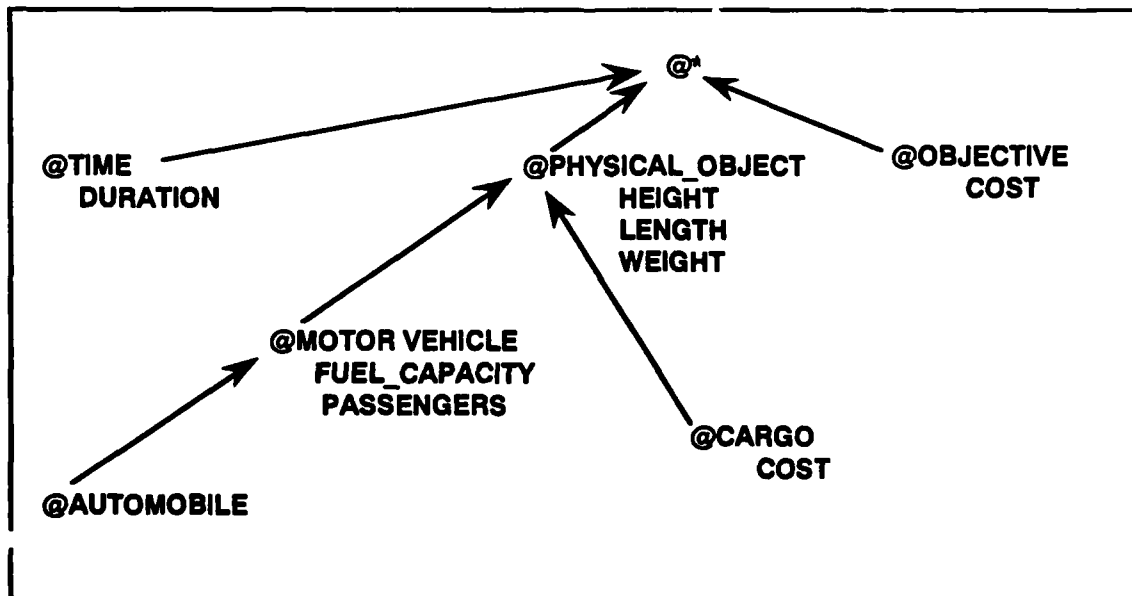


Figure 5.3 A Concept Graph

and @CARGO in the concept graph. In this case, both concepts inherit the WEIGHT quantity from @PHYSICAL_OBJECT. Consequently, the dimensional component determined for the sum "X + Y" is "WEIGHT of @PHYSICAL_OBJECT". If @AUTOMOBILE and @CARGO did not have a common generalization with the WEIGHT quantity, the type of "X + Y" would be "numerical type error".

When the concept graph is a tree, deciding whether two concepts have a common generalization is straightforward. First, navigate the directed path from one concept to the root node, marking the visited nodes, including the origin node, along the way. Next, navigate the directed path from the other concept to the root node. If a marked node is encountered before the root node, this intersection shows that the concepts have a common generalization. If no intersection occurs before the root node, the resultant is a numerical type error.

In our taxonomic structure, the existence of a common generalization is only a necessary condition for a concept coercion. A second condition must also be satisfied: the quantity labels which attribute each compared concept must also be assigned or inherited characteristics of the generalizing concept. If they are peculiar to each specialization, concept differences cannot be reconciled.

Determining whether this second condition is met is similar to determining whether the quantity of each concept specialization is equivalent. To determine quantity equivalence, the quantity canonical forms associated with each concept must be compared. This can be done, as we described in Chapter 4, by comparing the exponents of one quantity vector for equality with the exponents of the other quantity vector. To determine whether these same quantities are characteristics of the common concept generalization, each quantity vector must be compared to a new data structure, called a *scope vector*. The scope vector is a record of all quantity labels reachable along a directed path from a

particular concept node within the concept graph. The elements of the vector have the same order as the quantity array. Instead of storing exponents, each cell has an entry of "1" or "0." A "1" indicates that a particular quantity is reachable; a "0" indicates that the quantity is not reachable. Each quantity vector is compared to this record as follows: for each non-zero exponent in the quantity vector, the corresponding cell in the other array must have an entry of "1." If this condition is not satisfied, the generalized concept cannot be used as a coercion.

The disadvantage of tree-structured taxonomies is that they sometimes force us to duplicate information in order to maintain the acyclic structure. Figure 5.4 contains two concept graphs. The first concept graph is a tree that contains two concepts called "@FACTORY" and "@WAREHOUSE." The second concept graph is a revision of the tree structure that creates a new concept that has the assigned and inherited quantities of both @FACTORY and @WAREHOUSE, called "@FACTORY_THAT_IS_A_WAREHOUSE." The parent concept of this new concept is @FACTORY. Notice that it is necessary to restate all

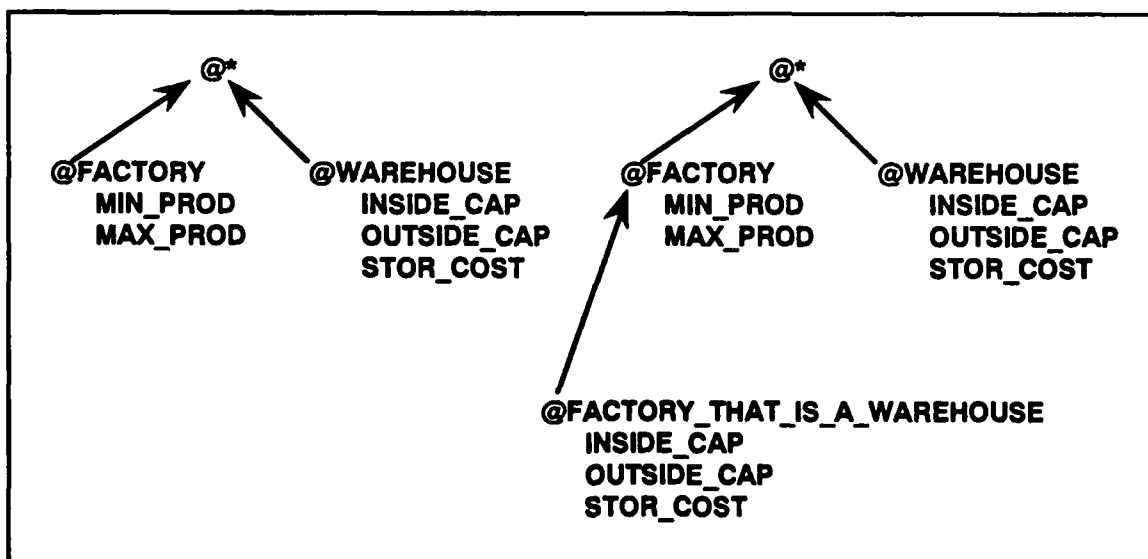


Figure 5.4 Concept Trees

of the @WAREHOUSE quantities (assigned and inherited) that FACTORY_THAT_IS_A_WAREHOUSE would not inherit from @FACTORY.

An alternative to this kind of duplication is to allow a concept to inherit from multiple superiors. While this solution is intuitively appealing, it complicates the reasoning needed to resolve concept mismatches in types. Under a multiple inheritance protocol, the concept graph has a more general structure. It is a rooted, acyclic digraph that contains many directed spanning trees.

When a concept graph has multiple directed spanning trees, the outcome of addition, subtraction or comparison of numerical objects of unequal types depends upon the spanning tree used to determine type equivalences. An upper bound on the number of potential alternatives can be computed by the following method (Tutte <1948>):

Let D be the in-degree matrix of a rooted digraph $G[V,E]$, where V is the set of nodes and E is the set of directed edges.

$$D(i,j) = \begin{cases} \text{the in-degree of node } i & \text{if } i = j; \text{ else} \\ -k, & \text{where } k \text{ is the number of edges in } G \text{ from } i \text{ to } j. \end{cases}$$

Theorem: The number of directed spanning trees with root r of a finite digraph with no self-loops is given by the determinant of the minor of its in-degree matrix which results from the elimination of the r th row and r th column.

The requirement that the quantities being compared between the two objects must be assigned or inherited traits of the common generalization provides one rule for selecting the appropriate spanning tree. Where more than one acceptable generalization exists, it may be necessary to enumerate all possible generalizations before an arithmetic expression is determined to be valid.

C. INDEXED CONCEPTS

When a single type statement is declared by the modeler for a set of symbolic numerical values, such as the "SHIPMENT" set of variables in Figure 5.5, all elements of the set share the same type. Although this notation for specifying types for sets of variables, parameters and functions is concise, it creates two security loopholes in the type system. First, siblings within the same set can be added, subtracted, or compared without restriction. For example, the arithmetic expression

SHIPMENT("DALLAS", "PITTSBURGH") - SHIPMENT("DETROIT", "MEMPHIS")

is type valid, even though this particular combination may be meaningless in the context of a transportation model.

The second loophole is that certain relationships between symbolic numerical values, such as parameters and variables, that are established by indexing schemes can be ignored without violating the type system. Consider the EML declarations given for the parameter set "SCOST" and the variable set "SHIPMENT" in Figure 5.5. Since the elements of both SCOST and SHIPMENT are discriminated by the same indexing scheme, the modeler's intention is that a unique SCOST parameter be associated with each member of the SHIPMENT variable set. For example, **SCOST("DALLAS","PITTSBURGH")** and **SHIPMENT("DALLAS", "PITTSBURGH")** are intended to be used together because their index suffixes are derived from the same set, "PATHS(i,j)," and have identical values. However, this distinction is not enforced by the type statements of SCOST and SHIPMENT. If **SCOST("DALLAS", "PITTSBURGH")** were multiplied by **SHIPMENT("DETROIT", "MEMPHIS")** in an EML function or constraint, the parallel type evaluation would yield

<< COST of @OBJECTIVE # US_\$ # >>.

ignoring the differences implied by different index labels.

```

<< QUANTITIES
  WEIGHT : LBS ;
  COST : US_$ ;
  DURATION: DAY ; >>

<< CONCEPTS
  @BUTTER [WEIGHT] ;
  @OBJECTIVE [COST] ;
  @TIME [DURATION] ; >>

SETS
  DAIRIES ; << nominal >>
  WAREHOUSES ; << nominal >>
  PATHS(i,j) := { CROSS ({DAIRIES} , {WAREHOUSES} ) } ;

VARIABLES

  SHIPMENT(i,j) {PATHS}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

  POSITIVE: SHIPMENT(i,j);

PARAMETERS
  SCOST(i,j) {PATHS}; << COST of @OBJECTIVE / (WEIGHT of @BUTTER
                                     / DURATION of @TIME) # US_$ / ([100] LBS / DAY) # >>

  SUPPLY(i) {DAIRIES}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

  DEMAND(j) {WAREHOUSES}; << WEIGHT of @BUTTER / DURATION of @TIME
                                     # [100] LBS / DAY # >>

FUNCTIONS
  OBJECTIVE := SUM (i,j) {PATHS} (SCOST(i,j)*SHIPMENT(i,j));
                                     << COST of @OBJECTIVE # US_$ # >>

CONSTRAINTS
  OUTBOUND(i) {DAIRIES} := SUM (j) {PATHS} (SHIPMENT(i,j)) =L= SUPPLY(i);
                                     << WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY # >>

  INBOUND(j) {WAREHOUSES} := SUM (i) {WAREHOUSES} (SHIPMENT(i,j))
                                     =E= DEMAND(j);
                                     << WEIGHT of @BUTTER / DURATION of @TIME # [100] LBS / DAY # >>

```

Figure 5.5 EML Schema With Typing

One way to eliminate these security breaches would be to identify the fundamental numerical objects in a model and provide a different type statement for each one. Then, polymorphic typing could be used to type objects whose types can be derived from these

declarations. For example, if "A(i,j)" is a technical coefficient in a linear constraint, "X(j)" is a decision variable and "B(i)" is the constraining resource, the type of A("FISHERMEN", "SARDINES") could be determined from the type declarations of X("SARDINES") and B("FISHERMEN"). There are two disadvantages to this proposal. It necessitates the enumeration of many individual type definitions, and it lacks the perspicuity of indexed EML notation.

A more concise and more powerful alternative is to use the modeler's indexing scheme to individualize types. This can be done by appending index suffixes to concepts declared in the type context. For example, each member of the "SHIPMENT" set in Figure 5.5 can be given a different type by changing the set type statement dimensional description from

WEIGHT of @BUTTER / DURATION of @TIME

to

WEIGHT of @BUTTER(i,j) / DURATION of @TIME.

This innovation also eliminates the other security loophole involving composition of indexed objects. For example, if SCOST("DALLAS", "PITTSBURGH") were multiplied by SHIPMENT ("DETROIT", "MEMPHIS") the type of the resultant would be

**<<COST of @OBJECTIVE*WEIGHT of @BUTTER("DETROIT","MEMPHIS")
/ WEIGHT of @BUTTER("DALLAS","PITTSBURGH") # US_\$ # >>.**

Any attempt to add the numerical resultant to another object which had the type of << COST of @OBJECTIVE # US_\$ # >> would generate a numerical type error.

Concept indexing can be used in other ways to enforce the intent of the modeler. It can be used to regulate the creation of additive aggregates within numerically valued sets. Consider the "OUTBOUND" and "INBOUND" constraint sets defined in Figure 5.5. If each SHIPMENT(i,j) variable has a different type because "@BUTTER" is indexed, then each of these constraints would generate a type error because they attempt to sum variables with

different types. Although we could resolve these differences by applying modeler-defined conversion factors, as described in Chapter 4, a more elegant solution is possible using indexed concepts and a concept graph.

Figure 5.6 is a concept graph for an improved model that includes indexed concepts. The upward pointing arrows follow the conventions of the last section: they are read as "is a specialization of." Notice that "@BUTTER(i,j)" has two parent concepts, "@BUTTER(i,.)" and "@BUTTER(.,j)". Both @BUTTER(i,.) and @BUTTER(.,j) possess the quality "WEIGHT" and pass it as a legacy to @BUTTER(i,j).

@BUTTER(i,.) is defined to represent butter that originates at one specific dairy "i" and terminates at any warehouse "j" in the "WAREHOUSES" index set. The dot notation, ".", is used to suppress the identity of the "j" and thus, capture the modeler's intent that the identity of the warehouse is not important for this concept. Similarly, "@BUTTER(.,j)" is defined to represent butter that originates at any dairy "i" and terminates at a specific warehouse "j". These two concepts are used to capture the modeler's intent that it is only valid to sum shipments that originate at the same dairy "i" or terminate at the same warehouse "j". All other shipment aggregations are invalid. For example, an arithmetic

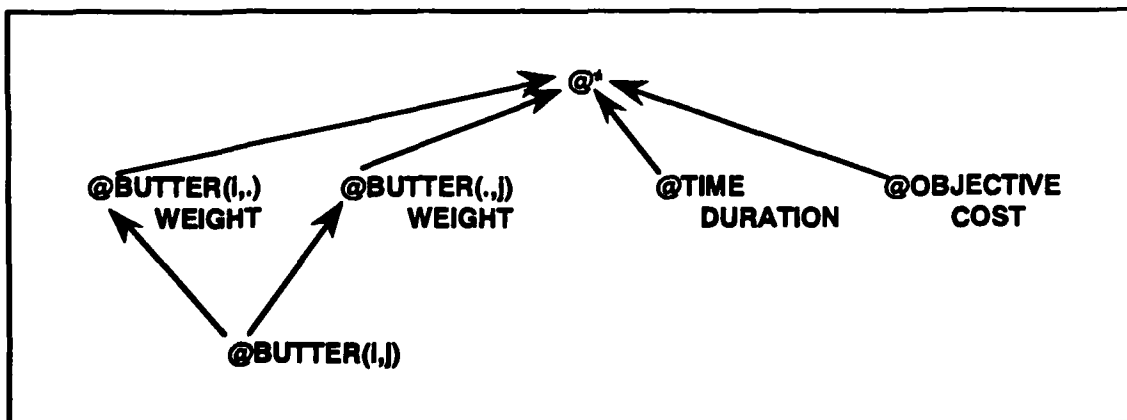


Figure 5.6 Concept Graph With Indexed Concepts

expression such as

SHIPMENT("DALLAS","DENVER") + SHIPMENT("DALLAS","WICHITA")

would produce a valid type of

<<WEIGHT of @BUTTER("DALLAS", ".") / DURATION of TIME# LBS / DAY#>> .

However, an expression such as

SUM (I,J) {PATHS} (SHIPMENTS(I,J))

would not be a legitimate construct in the type context in Figure 5.5 because @BUTTER(I,J) does not have "@BUTTER(...)" as an ancestor. Although this graph employs multiple inheritance, the "SUM" operation that uses its generalizations is index specific, eliminating any ambiguity. Figure 5.7 is a revision of Figure 5.5 using indexed concepts.

A third use of concept indexing is to regulate the comparison of additive aggregates between different numerical sets that share one or more simple indices. To demonstrate this feature, we extend our improved model (Figure 5.7) in the following way. Shipments of butter which arrive at warehouses are used to satisfy local inventory requirements. Shipments received in excess of local needs are forwarded at some expense to consumers. The additional EML and type language declarations for this embellishment are shown in Figure 5.8. The concept graph for the complete model (Figures 5.7 & 5.8) is shown in Figure 5.9.

The type context now includes three additional indexed concepts: "@BUTTER(j,k)," "@BUTTER(.,k)" and "@BUTTER(j,.)." These objects represent butter shipped from a specific warehouse to a specific consumer, butter shipped from any warehouse to a particular consumer, and butter shipped from a particular warehouse to any consumer, respectively. While these concepts and the ones that were introduced earlier are adequate to regulate the aggregation of shipments between dairies and warehouses and between warehouses and customers, they are not sufficient to model shipments which enter and

leave the same warehouse. Each material balance constraint, "BALANCE()", is invalid because its aggregated inbound shipments and aggregated outbound shipments have

```

<< QUANTITIES
  WEIGHT : LBS ;
  COST : US $ ;
  DURATION: DAY ; >>

<< CONCEPTS
  @ <-- @BUTTER(i,.) [WEIGHT] ;
  @ <-- @BUTTER(.,j) [WEIGHT] ;
  @BUTTER(i,.) <-- @BUTTER(i,j)
  @BUTTER(.,j) <-- @BUTTER(i,j)>>

SETS
  DAIRIES i; << nominal >>
  WAREHOUSES j; << nominal >>
  PATHS(i,j) := { CROSS ({DAIRIES} , {WAREHOUSES} ) };

VARIABLES
  SHIPMENT(i,j) {PATHS}; << WEIGHT of @BUTTER(i,j) / DURATION of @TIME
                           # [100] LBS / DAY # >>
  POSITIVE SHIPMENT(i,j);

PARAMETERS
  SCOST(i,j) {PATHS}; << COST of @OBJECTIVE / (WEIGHT of @BUTTER(i,j)
                           / DURATION of @TIME) # US_$ / ([100] LBS / DAY) # >>

  SUPPLY(i) {DAIRIES}; << WEIGHT of @BUTTER(i,.) / DURATION of @TIME
                           # [100] LBS / DAY # >>

  DEMAND(j) {WAREHOUSES}; << WEIGHT of @BUTTER(.,j) / DURATION of @TIME
                           # [100] LBS / DAY # >>

FUNCTIONS
  OBJECTIVE := SUM (i,j) {PATHS} (SCOST(i,j)*SHIPMENT(i,j));
                           << COST of @OBJECTIVE # US_$ # >>

CONSTRAINTS
  OUTBOUND(i) {DAIRIES} := SUM (j) {PATHS} (SHIPMENT(i,j)) =L= SUPPLY(i);
                           << WEIGHT of @BUTTER(i,.) / DURATION of @TIME # [100] LBS / DAY # >>

  INBOUND(j) {WAREHOUSES} := SUM (i) {WAREHOUSES} (SHIPMENT(i,j))
                           =E= DEMAND(j);
                           << WEIGHT of @BUTTER(.,j) / DURATION of @TIME # [100] LBS / DAY # >>

```

Figure 5.7 EML Schema Revision With Indexed Typing

different dimensional descriptions. The left-hand side of **BALANCE()** has the type

<< WEIGHT of @BUTTER(.,j) / DURATION of @TIME # LBS / DAY # >>.

The right-hand side of **BALANCE()** has the type

<< WEIGHT of @BUTTER(j,.) / DURATION of @TIME # LBS / DAY # >>.

<< CONCEPT GRAPH

@* <-- @BUTTER(*,j)[WEIGHT]
@BUTTER(*,j) <-- @BUTTER(i,j)
@BUTTER(*,j) <-- @BUTTER(j,k) >>

SETS

CUSTOMERS k; << nominal >>
PATHS(i,j) := { CROSS ({WAREHOUSES} , {CUSTOMERS}) };

VARIABLES

FLOW(j,k) {LINKS}; << WEIGHT of @BUTTER(j,k) / DURATION of @TIME
[100] LBS / DAY # >>

POSITIVE: FLOW(j,k);

PARAMETERS

COST(i,j) {LINKS}; << COST of @OBJECTIVE / (WEIGHT of @BUTTER(j,k)
/ DURATION of @TIME) # US_\$ / ([100] LBS / DAY) # >>

RETAIL(i) {DAIRIES}; << WEIGHT of @BUTTER(.,k) / DURATION of @TIME
[100] LBS / DAY # >>

DEMAND(j) {WAREHOUSES}; << WEIGHT of @BUTTER / DURATION of @TIME
[100] LBS / DAY # >>

FUNCTIONS

TOTAL_COST := SUM (j,k) {LINKS} (COST(j,k)*FLOW(j,k)) + OBJECTIVE;
<< COST of @OBJECTIVE # US_\$ # >>

CONSTRAINTS

BALANCE(i) {WAREHOUSES} := SUM (j) {PATHS} (SHIPMENT(i,j))
=L= SUM (k) {LINKS} (FLOW(j,k)) - DEMAND(j) ;
<< WEIGHT of @BUTTER(*,j) / DURATION of @TIME # [100] LBS / DAY # >>

Figure 5.8 Transshipment Model Components

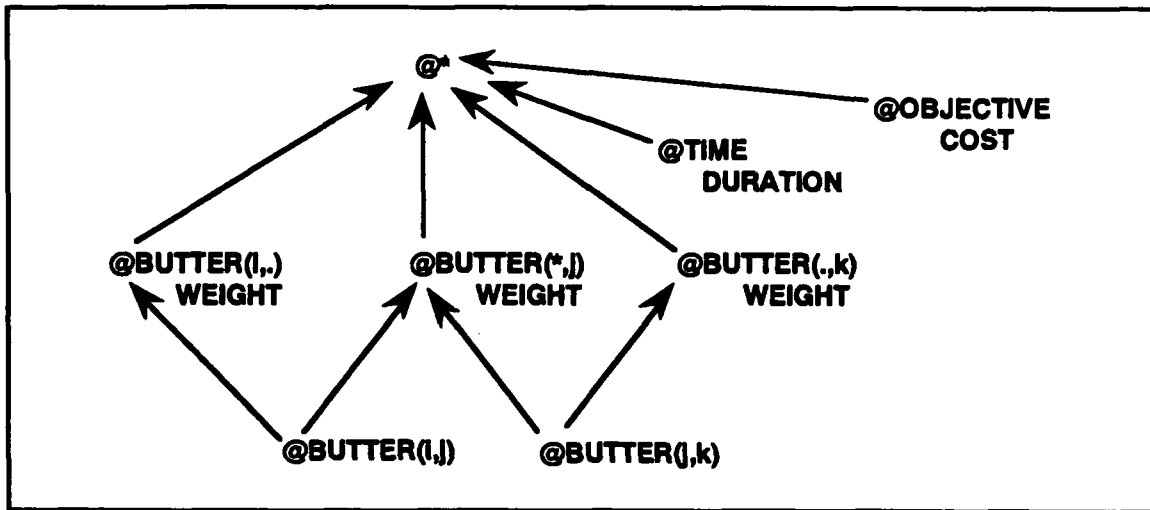


Figure 5.9 Concept Graph For Transshipment Model Schema

The source of the problem is that $@BUTTER(*,j)$ and $@BUTTER(j,*)$ do not have the same index suffixes. The origin of the suffix of $@BUTTER(*,j)$ is the "PATHS(i,j)" index set. The origin of the suffix of $@BUTTER(j,*)$ is the "LINK(j,k)" index set. In both cases, however, only the "j" or warehouse index is visible, the other index in each suffix has been suppressed. If we adopt the convention that indices are not dummy, that is, an index is uniquely associated with a set of identifiers, we can resolve this problem by concluding that the order in which "j" appears in a suffix is immaterial. Thus, "(j,*)" or "(*,j)" are indistinguishable. Figure 5.9 introduces a new concept, " $@BUTTER(*,j)$," in the concept graph to document this interpretation.

The "*" operator is a powerful idea for manipulating concepts, but it should be used cautiously. Indiscriminate use would dilute the power of typing.

D. APPLICATION DOMAINS

The type language, in its current form, has a very low degree of semantic commitment: while the language distinguishes between unit labels, quantity labels and concept labels and regulates how these labels may be composed to form language

statements, individual labels have no predetermined meaning. For example, if a modeler wishes to ascribe the dimension of length to a modeling language parameter, he could represent this quantity with an arbitrary label. The labels "LNGTH" or "L" might be chosen as suggestive mnemonics for the word "length," however, "ABC123" would be equally legitimate. The purpose of this section is to discuss the advantages that accrue from fixing the meaning of labels in the type language.

Standardization of labels for units, for quantities and for concepts increases the value of typing as model documentation. This allows potential users to understand what the model purports to do without learning to decipher the model-peculiar mnemonics used to name parameters, variables and functions. Related to the issue of facilitating communication between people is the issue of coordinating models with the information resources that provide their data. Label standardization could enable a parameter's type to be a formal specification to an exogenous data source, such as a database. This would provide additional model security by preventing integrity problems which occur when the data source changes but these changes are not reflected in the model.

Other benefits flow from fixing the meaning of labels. At present, if the modeler wishes to use the metric units of length, "meter" and "centimeter," to describe objects in a model, and foresees a need to coerce one unit to the other, the necessary identity must be included in the model. In general, the modeler must enumerate many of the unit, quantity, and concept coercions required. If these labels are standardized, the identities that relate units to other units, quantities to other quantities and concepts to other concepts can become an exogenous part of the model. For example, the Avoirdupois Weight system is applicable in many different modeling settings. The modeler, or someone else, could prepare a list of identities that capture all commensurate relationships between "ounces", "pounds", "stones", "short tons", and "long tons." These identities could be stored in a

separate text file and be made available during syntactic analysis. Besides the obvious advantages of reuse and a more concise representation, this approach also makes the model more robust. The modeler is no longer obliged to anticipate the coercions that may be necessary as the model and the data evolve.

If quantity labels and concept labels are standardized, concept graphs may also be standardized and reused. The advantage of reusing concept graphs is that it enforces consistency in the way that types derived from standard labels are used in other models. For example, the concept graph defined for a transportation model in the last section (Figure 5.8) did not allow flows with different origins and different destinations to be summed. This idea is developed further in a subsequent chapter on integrated modeling.

Taken collectively, each of the above innovations can be used to define *domains* for different modeling applications. Each domain would consist of a closed set of unit labels, quantity labels, and concept labels; their identities; and a concept graph. The idea is that a *master modeler* who is expert in modeling a particular application, like banking or the manufacture of ammunition, would identify the concepts, quantities and units that are central to his area of expertise. He would then dictate identities and a concept graph that would be general enough to cover most of the valid models that could be built for the application, but limited enough that violation of any important property or relationship would be caught by the type analyzer. The domain would then be made available to a person modeling a specific problem within the application area.

Domains are potentially useful in both industrial and academic settings. Dolk and Konsynski <1985> observe that the advent of personal computers and sophisticated modeling software have "put models and modeling capabilities into the hands of unskilled modelers." In their view, the increases in personal productivity attributable to decentralized modeling are being offset by organizational turmoil caused by lack of control

of this function. For example, lack of standardization in assumptions and in data specifications raise questions of model accuracy, validity and integrity. Domains could provide part of the solution to these problems by providing a uniform way to view models within an organization and a way to enforce that view through the type system.

The utility of domains is not limited to modeling application. Domains could also be used to help teach modeling skills. A teacher could prepare a domain to codify the important properties of a class of models. A class assignment might be to formulate a typed modeling language schema for a particular textbook problem using that domain. The student would get immediate feedback if a proposed solution violated any of the important teaching points in the exercise. In effect, the type analyzer and the domain would act as a surrogate for the teacher as it looks over the shoulder of the student and points out errors.

VI. TYPING APPLIED TO INTEGRATED MODELING

The purpose of this chapter is to discuss the application of typing to a method of modeling called *integrated modeling*. We define an integrated model to be a model that is synthesized from two or more distinct, but logically related models. The chapter is presented in three sections. We begin by describing a context for the practice of integrated modeling and summarizing the techniques used by modelers to unify models expressed as mathematical programs. In the second section we provide two sets of examples. The first set illustrates integrated modeling techniques in an untyped modeling language and serves as a basis for discussing the support provided by untyped modeling languages to integrated modeling. The second set demonstrates the support that a typed modeling language could provide to a representative subset of the same model integrations. The last section of the chapter introduces a new method of constructing an *integrated model* based upon what we call a *library unit*. A library unit is a model or model fragment that includes a typed model schema, a concept graph, a unit system and an interface that regulates its use. A syntax is proposed for a model integration language utilizing library units. Finally, examples are provided that illustrate both the language and the library unit construct.

A. INTEGRATED MODELING

1. A Rationale For Building Models of Systems As Integrated Models

An area of growing importance in OR/MS is the modeling and optimization of problems involving the operations of systems. We define a system as a group of objects or processes, interrelated in a regular way to form a complex whole. While there have been many successful applications of OR/MS techniques to diverse segments of systems, successful studies that model and optimize both the components and the interconnections in

a system have been more rare. However, as optimization technology has advanced, instances of system models have been solved with increasing frequency. During the past twenty years successful results have been reported for the following classes of problems: vehicle routing/inventory allocation (Federgruen and Zipkin <1984>), production/finance/marketing (Damon and Schramm <1972>); production/distribution (Brown, Graves and Honczarenko <1987>, (Cohen and Lee <1988>) and investment/finance (McInnes and Carlton <1982>).

Problems which involve the operation of real-world systems challenge our ability to formulate and implement models as well as our arsenal of algorithms. It is generally accepted that people seem to be able to keep only a few distinct things in their minds at one time. Consequently, the complexity of understanding any system increases rapidly with its size. As the complexity of a task increases, the potential for error increases disproportionately. As modeling errors increase, model validation consumes a disproportionate share of the available modeling resources.

A common approach to dealing with a large problem of any kind is to decompose it into smaller problems. This divide and conquer strategy is the rationale for the modeling technique called *integrated modeling*. Integrated models are built in a modular fashion, unifying independent, but logically connected models. Each component model represents some coherent aspect of the modeled system. By developing a model of a complex system as a set of independent, smaller models, the total difficulty of the design task decreases.

Integrated models of systems possess another desirable quality. If the component models are already validated, validating the system model is considerably easier. Only the component model interconnections need to be tested and sanctioned. This fact is particularly appealing when we consider that organizations that have practiced modeling for some time already possess validated models for various aspects of their activities.

2 . The Mechanics of Integrating Algebraic Models

In this section we present integrated modeling from a mechanical perspective. We do not presume to explain why a modeler chooses to decompose a system model in a particular way nor do we advocate a methodology for this task. We do know, however, that the possible ways of integrating two models are limited in two ways. First, they are restricted by the grammar of algebraic notation. Second, there are only two possible configurations that can be formed from two model components. Either they preserve their independence and are interconnected through the definition of new constraints, or one model is subordinated to the other by adopting the naming and indexing conventions for parameters and/or variables in the dominant model.

When two models expressed in algebraic notation are integrated, the modeler binds them together in one or more of the following ways:

- (1) by replacement: the parameters, variables or indices in one model are replaced by the parameters, variables or indices in the other model;
- (2) by index composition: a Cartesian product of an index set from one model is taken with an index set from the other model. New parameters and/or variables are then defined over the new index set;
- (3) by constraint composition: new constraints are created by composing the left-hand side (right-hand side) of a constraint in one model with the left-hand side (right-hand side) of the other model by a relational operator;
- (4) by objective function composition: a new objective function is created by adding or subtracting the objective functions of the component models;
- (5) by functional definition: the value of parameters (variables) in one model are defined to be a mathematical function of parameters (variables or parameters and variables) in the other model; and,
- (6) by constraint creation: a new constraint is defined which relates an arithmetic expression of one or more variables in one model to an arithmetic expression of one or more variables in the other model.

In addition to these six techniques, the implementor of an integrated model has the full power of algebraic notation at his disposal. New variables and new parameters may be defined over existing index sets. Completely new index sets (with their associated variables and parameters), new variables, new parameters and new constraints may be defined. The modeler is also free to eliminate symbols and constraints from the component models as necessary.

Untyped and typed EML schemas are introduced throughout the remainder of this chapter to illustrate the mechanics of integrated modeling and to demonstrate typing. Each schema employs one or more of the following graphical conventions: boldfaced italics, double line boxes and strikeout (- - -). Boldfaced italics are used to highlight new constructs in schemas. Double line boxes are used in schemas of integrated models to identify the constructs contributed by one of the two component models. Strikeout is used in schemas of integrated models to eliminate superfluous constructs.

B. INTEGRATING ALGEBRAIC MODELS WITH UNTYPED AND TYPED MODELING LANGUAGES

1. Integrated Modeling In An Untyped Language

In this section we construct five integrated models using transportation models and a production/location model as components. The purpose of these examples is to demonstrate some of the techniques we have described in the previous section and to point out the strengths and weaknesses of untyped modeling languages in this endeavor. The component models we will use are listed in Figures 6.1, 6.2, and 6.3 and will be referred to as Model_1, Model_2 and Model_3, respectively. Each one is written in EML (Appendix A).

```

SETS
  SOURCE I ;
  SINK J ;
  ARC(I,J) := { CROSS( {SOURCE} , {SINK} ) };

VARIABLES
  X(I,J) {ARC} ;
  POSITIVE: X(I,J);

PARAMETERS
  S(I) {SOURCE} ;
  D(J) {SINK} ;
  C(I,J) {ARC} ;

FUNCTIONS
  OBJ := SUM(I,J) {ARC} ( C(I,J)*X(I,J) ) ;

CONSTRAINTS
  SUPPLY(I) {SOURCE} := SUM(J) {ARC} ( X(I,J) ) =L= S(I) ;
  DEMAND(J) {SINK} := SUM(I) {ARC} ( X(I,J) ) =E= D(J) ;

```

Figure 6.1 Model_1 (Transportation Model as EML Schema)

```

SETS
  ORIGIN k ;
  DESTINATION l ;
  LINK(k,l) := {CROSS( {ORIGIN} , {DESTINATION} ) };

VARIABLES
  Y(k,l) {LINK} ;
  POSITIVE: Y(k,l);

PARAMETERS
  SUP(k) {ORIGIN} ;
  DEM(l) {DESTINATION} ;
  COST(k,l) {LINK} ;

FUNCTIONS
  TOT_$ := SUM(k,l) {LINK} ( COST(k,l)*Y(k,l) ) ;

CONSTRAINTS
  OUT_BND(k) {ORIGIN} := SUM(l) {LINK} ( Y(k,l) ) =L= SUP(k) ;
  IN_BND(l) {DESTINATION} := SUM(k) {LINK} ( Y(k,l) ) =E= DEM(l) ;

```

Figure 6.2 Model_2 (Transportation Model as EML Schema)

a. Example 1

The simplest integrated model that can be formed from Model_1 and Model_2 is to sum their two objective functions (technique 4) and to take the union of their constraint sets. The composite objective function would be written in EML as

$$\text{TOTAL_COST} := \text{TOT_\$} + \text{OBJ} ;$$

This new model could be used to represent the decisions and resulting costs of operating two independent distribution systems (although it would be preferable to optimize each component model separately).

b. Example 2

Three other integrations of Model_1 and Model_2 are apparent when both models are considered pictorially. One integration would be to connect the sink nodes in Model_1's graph to the source nodes in Model_2's graph with arcs (Figure 6.4). One possible use for such a model would be to represent a system where the things shipped from "SOURCE i" to "SINK j" in Model_1 were forwarded at some cost to "ORIGIN k" in Model_2.

An algebraic representation of this graphical model (Figure 6.5) is achieved by adding several interconnections to the component models. First, using technique 1, we define a new index set, "PORTAGE", which is the Cartesian product of the SINK and ORIGIN index sets defined in Model_1 and Model_2, respectively. New nonnegative variables, "P(j,k)", and new parameters, "CP(j,k)" are defined over the "PORTAGE" set to represent the level of flow and the cost per unit of flow between each SINK "j" and each ORIGIN "k." Next, a composite objective function, "TOTAL_COST" is defined (technique 4). TOTAL_COST is composed of the objective functions of Model_1 and Model_2 and a new term,

$$\text{SUM}(j,k) \{ \text{PORTAGE} \} \quad (\text{CP}(j,k) * \text{P}(j,k)).$$

```

SETS
  FACILITIES j ;
  PLANT k ;
  ACTIVITY (j,k) := { SELECT( CROSS( {FACILITY} , {PLANT} ) ) } ;

PARAMETERS
  CORP_DEMAND ;
  PLANT_COST(k) {PLANT} ;
  FAC_LIMIT_LOWER(k) {PLANT} ;
  FAC_LIMIT_UPPER(k) {PLANT} ;
  PROD_COST(j,k) {ACTIVITY} ;
  FAC_COST(j,k) {ACTIVITY} ;
  FAC_UTIL_LOWER(j,k) {ACTIVITY} ;
  FAC_UTIL_UPPER(j,k) {ACTIVITY} ;

VARIABLES
  OPEN_PLANT(k) {PLANT} ;
  PROD(j,k) {ACTIVITY} ;
  OPEN_FAC(j,k) {ACTIVITY} ;
  POSITIVE: PROD(j,k);
  BOOLEAN: OPEN_FAC(j,k), OPEN_PLANT(k);

FUNCTIONS

  OBJECTIVE := SUM(j,k) {ACTIVITY} (PROD_COST(j,k) * PROD(j,k) + FAC_COST(j,k) *
  OPEN_FAC(j,k))
  + SUM(k) {PLANT} (PLANT_COST(k) * OPEN_PLANT(k)) ;

CONSTRAINTS

  QUOTA := SUM(j,k) {ACTIVITY} (PROD(j,k)) =E= CORP_DEMAND ;
  CO_LOCATION(k) {PLANT} := FAC_LIMIT_LOWER(k) * OPEN_PLANT(k)
  =L= SUM(j) {ACTIVITY} (OPEN_FAC(j,k))
  =L= FAC_LIMIT_UPPER(k)*OPEN_PLANT(k) ;

  UTILIZATION(j,k) {ACTIVITY} := FAC_UTIL_LOWER(j,k) * OPEN_FAC(j,k)
  =L= (PROD(j,k))
  =L= FAC_UTIL_UPPER(j,k)*OPEN_FAC(j,k) ;

  FAC_ALLOCATION(j) {FACILITIES} := SUM(k) {ACTIVITY} (OPEN_FAC(j,k))
  =L= 1 ;

```

Figure 6.3 Model_3 (Production Model as EML Schema)

The new term represents the cost contribution of items shipped from each SINK "j" to each ORIGIN "k." The last interconnection is created by creating two new sets of constraints that enforce a material balance across each SINK "j" and across each ORIGIN "k."

"MAT_BAL1(j)" consists of the left-hand side of the "DEMAND(j)" constraint in Model_1 set equal to a sum of the new variables,

$$\text{SUM}(k) \{ \text{PORTAGE} \} (P(j,k)).$$

"MAT_BAL2(k)" consists of the left-hand side of the "SUPPLY(k)" constraint in Model_2 set equal to a different sum of the new variables,

$$\text{SUM}(j) \{ \text{PORTAGE} \} (P(j,k)).$$

The last step in the development of the model in Figure 6.5 from Model_1 and Model_2 is to eliminate the component-contributed constraints which do not apply to the integrated formulation. Parameters and variables which are defined but not referenced in the integrated model are also removed. The "D(j)" parameters and the DEMAND(j) constraints contributed by Model_1, and the "SUP(k)" parameters and the "OUT_BND(k)" constraints contributed by Model_2 are eliminated to enforce simple throughput restrictions on SINK "j" and ORIGIN "k," respectively.

c. Example 3

Another way of integrating two transportation models graphically is to superimpose the sink nodes in one model's graph upon the source nodes in the other model's graph (Figure 6.6). The physical analog of such a graph would be a distribution

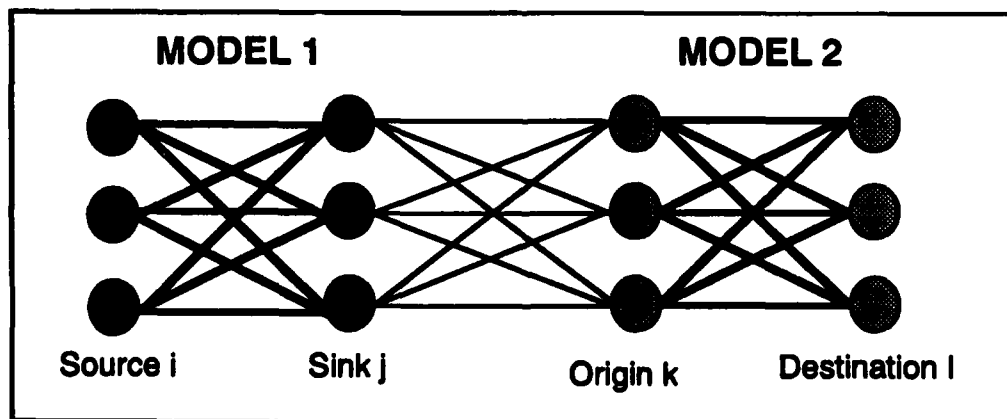


Figure 6.4 Integration By Connecting Arcs


```

SOURCE i;
SINK j;
ARC(i,j) := {SOURCE} x {SINK};

```

```

ORIGIN k ;
DESTINATION l ;
LINK(k,l) := { CROSS( {ORIGIN} , {DESTINATION} ) };

```

VARIABLES

X(i,l) {ARC}; POSITIVE: X(i,l);
P(j,k) {PORTAGE}; POSITIVE: P(j,k);
Y(k,l) {LINK}; POSITIVE: Y(k,l);

```
S(I) {SOURCE};  
D(I) {SINK};  
C(I,J) {ARC};
```

SUP(k) {ORIGIN};
DEM(l) {DESTINATION};
COST(k,l) {LINK};

OBJ := SUM(i,j) {ARC} (C(i,j)*X(i,j)) ;

$$\text{TOT } \$:= \text{SUM}(k,l) \{ \text{LINK} \} (\text{COST}(k,l) * Y(k,l)) ;$$

TOTAL_COST := OBJ + TOT_\$ + SUM (j,k) {PORTAGE}(CP(j,k)*P(j,k));

$$\text{SUPPLY}(i) \text{ (SOURCE)} := \text{SUM}(j) \text{ (ARC)} (X(i,j)) = L = S(i);$$

$$\text{DEMAND}(j) \text{ (SINK)} := \text{SUM}(i) \text{ (ARC)} (X(i,j)) = E = D(j);$$
$$\begin{aligned} \text{OUT_BND}(k) \{\text{ORIGIN}\} &:= \text{SUM}(l) \{\text{LINK}\} (Y(k,l)) - L - \text{SUP}(k); \\ \text{IN_BND}(l) \{\text{DESTINATION}\} &:= \text{SUM}(k) \{\text{LINK}\} (Y(k,l)) - E - \text{DEM}(l); \end{aligned}$$

```

MAT_BAL_1 {SINK} := SUM(I) {ARC} (X(I,J))
                  =E= SUM(k) {PORTAGE} (P(J,k)) ;
MAT_BAL_2 {ORIGIN} := SUM(J) {PORTAGE} (P(J,k))
                  =E= SUM(I) {LINK} (Y(k,I)) ;

```

97

system in which transshipment points would correspond to the superimposed nodes. An algebraic representation of this integrated model is shown in Figure 6.7. In this formulation, Model_2 is subordinate to Model_1. Technique 1 has been applied, replacing Model_2's ORIGIN "k" index set with Model_1's SINK "j" index set. In conjunction with this change, all references to ORIGIN "k" from sets, parameters and variables contributed by Model_2 to the integrated model are redirected to SINK "j."

After index usage has been redefined, composition of objective functions (technique 4) and constraint composition (technique 5) are applied. As was done in Example 1, the objective functions of Model_1 and Model_2 are summed to define an objective function for the integrated model (TOTAL_COST). The "MAT_BAL(j)" constraint is an equality composed of the left-hand side of the DEMAND(j) constraint from Model_1 and the left-hand side of the "OUT_BND(j)" constraint from Model_2. Notice that index of "OUT_BND" has been changed from "k" to "j" to conform with the index replacement done at the beginning of this example.

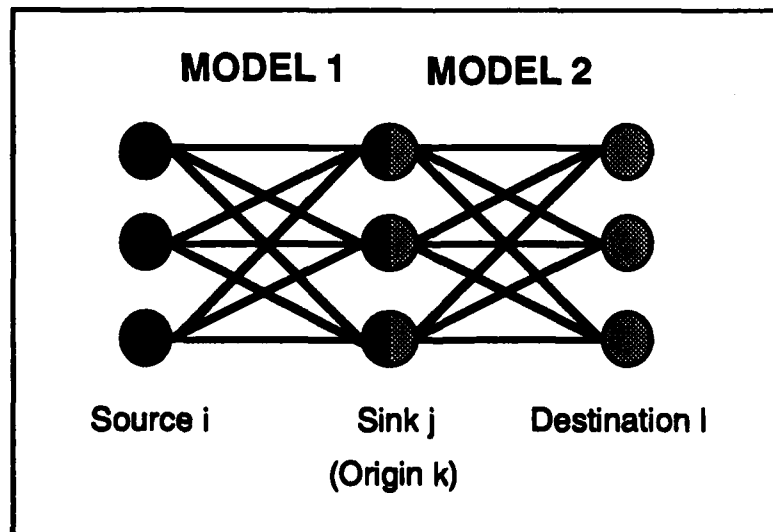


Figure 6.6 Integration By Superimposing Source Nodes on Sink Nodes

The last modeling step in this integration is to eliminate the `OUT_BND()` constraint and the `SUP()` parameter contributed by `Model_2`. This was also done in Example 2 and the justification for it is the same.

d. Example 4

The last integration of two transportation models we consider is the superposition of sink nodes and the superposition of source nodes to create a multi-commodity transportation model (Figure 6.8). We have made an arbitrary decision in the algebraic representation of this model (Figure 6.9) for `Model_1` to dominate `Model_2`. `SOURCE "i"` has replaced `ORIGIN "k"`, `SINK "j"` has replaced `DESTINATION "l"` and `"ARC(i,j)"` has replaced `"LINK(k,l)"`.

Two other interconnections have been added to bind the component models together. Using technique 6, a joint capacitation constraint, labelled `"CAPACITY(i,j)"` in Figure 6.9, has been defined. Each `CAPACITY(i,j)` constraint is an inequality. The left-hand side of the constraint consists of a sum of the decision variables in each model component that reference the same `ARC(i,j)` element. The right-hand side of the inequality is a new parameter, `"UPPER_BOUND(i,j)"`, that represents the upper bound on total flow across `ARC(i,j)`. The last interconnection is the composite objective function, `"TOTAL_COST."`

e. Example 5

The previous four examples have all involved integrating two specific models within the class of transportation models. We now consider the integration of `Model_2`, a transportation model, to `Model_3`, a mixed-integer formulation of a production/location model.

`Model_3` (Figure 6.3) is a simplification of a multi-commodity model developed and solved by Brown, Graves and Honczarenko <1987> for Nabisco Brands.

SETS

SOURCE I;
 SINK J;
 ARC(I,J) := {SOURCE} x {SINK};

ORIGIN K;
 DESTINATION I;
 LINK(J,I) := { CROSS({SINK} ,{DESTINATION})};

VARIABLES

X(I,J) {ARC}; POSITIVE: X(I,J);

Y(J,I) {LINK}; POSITIVE: Y(J,I);

PARAMETERS

S(I) {SOURCE};
 D(I) {SINK};
 C(I,J) {ARC};

SUP(J) {SINK};
 DEM(I) {DESTINATION};
 COST(J,I) {LINK};

FUNCTIONS

OBJ := SUM(I,J) {ARC} (C(I,J)*X(I,J));

TOT_\$:= SUM(J,I) {LINK} (COST(J,I)*Y(J,I));

TOTAL_COST := OBJ + TOT_\$;

CONSTRAINTS

SUPPLY(I) {SOURCE} := SUM(I) {ARC} (X(I,J)) = L = S(I) ;
 DEMAND(I) {SINK} := SUM(I) {ARC} (X(I,J)) = E = D(I) ;

OUT_BND(J) {SINK} := SUM(I) {LINK} (Y(J,I)) = L = SUP(J) ;
IN_BND(I) {DESTINATION} := SUM(I) {LINK} (Y(J,I)) = E = DEM(I) ;

MAT_BAL(I) {SINK} := SUM(I) {ARC} (X(I,J)) = E = SUM (I) {LINK} (Y(J,I)) ;

Figure 6.7 EML Schema for Example 3

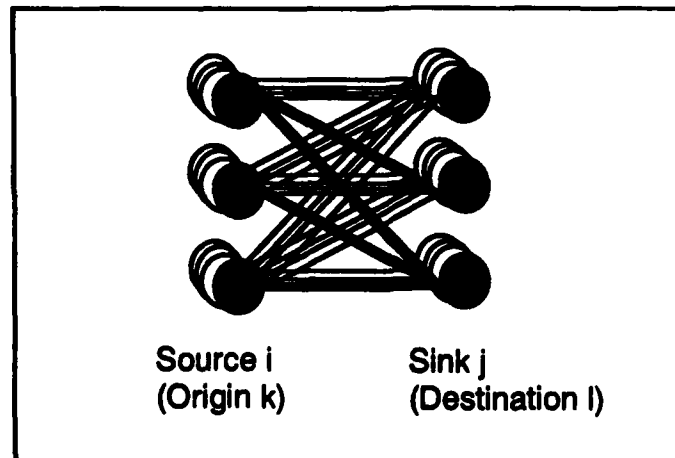


Figure 6.8 Integration By Superimposing All Node Sets

The original model is intended to manage complex problems involving plant site selection, equipment location and utilization and material distribution. Our version is restricted to one commodity and does not model secondary equipment nor the distribution of finished goods.

Before discussing how Model_2 and Model_3 are combined to form a more comprehensive model, an explanation of Model_3's notation and structure is necessary. The model introduces two indices and one derived set.

j is the index of FACILITIES (e.g., ovens)

k is the index of PLANT sites

ACTIVITY is a set containing the allowable combinations of facilities and plant sites. It is a subset of the Cartesian product of FACILITIES and PLANT.

The given data for the model are:

CORP_DEMAND corporate production requirement

PLANT_COST(k) the fixed cost of operating a plant at site " k "

**FAC_LIM_LOWER(k),
FAC_LIM_UPPER(k)** the minimum and maximum number of facilities at plant site " k "

SETS

SOURCE I;
SINK J;
ARC(I,J) := { CROSS({SOURCE}, {SINK}) };

ORIGIN K;
DESTINATION L;
LINK(K,L) := { CROSS({ORIGIN}, {DESTINATION}) };

VARIABLES

X(I,J) {ARC}; POSITIVE: X(I,J);

Y(I,J) {ARC}; POSITIVE: Y(I,J);

PARAMETERS

S(I) {SOURCE};
D(J) {SINK};
C(I,J) {ARC};

SUP(I) {SOURCE};
DEM(J) {SINK};
COST(I,J) {ARC};

UPPER_BOUND(I,J) {ARC};

FUNCTIONS

OBJ := SUM(I,J) {ARC} (C(I,J)*X(I,J));

TOT \$:= SUM(I,J) {ARC} (COST(I,J)*Y(I,J));

TOTAL_COST := OBJ + TOT_\$;

CONSTRAINTS

SUPPLY(I) {SOURCE} := SUM(J) {ARC} (X(I,J)) =L= S(I) ;
DEMAND(J) {SINK} := SUM(I) {ARC} (X(I,J)) =E= D(J) ;

OUT_BND(I) {SOURCE} := SUM(J) {ARC} (Y(I,J)) =L= SUP(I) ;
IN_BND(J) {SINK} := SUM(I) {ARC} (Y(I,J)) =E= DEM(J) ;

CAPACITY(I,J) {ARC} := X(I,J) + Y(I,J) =L= UPPER_BOUND(I,J);

Figure 6.9 EML Schema for Example 4

PROD_COST(j,k)	the cost of producing one unit on facility j at plant site "k"
FAC_COST(j,k)	the fixed cost of operating facility j at plant site "k"
FAC_UTIL_LOWER(j,k), FAC_UTIL_UPPER(j,k)	the minimum and maximum number of units that can be produced on facility "j" at plant site "k"

The decision variables for the model are:

OPEN_PLANT(k)	a 0-1 close-open variable for plant site "k"
OPEN_FAC(j,k)	a 0-1 assignment variable of facility "j" to plant site "k"
PROD(j,k)	the number of units produced on facility "j" at plant site "k"

The purpose of the mathematical program is to minimize "OBJECTIVE" subject to four sets of constraints: "QUOTA", "CO-LOCATION", "UTILIZATION" and "FAC_ALLOCATION." The QUOTA constraint assures that the corporate production requirement is met. The CO-LOCATION constraints limit the assignment of facilities to plant sites. The UTILIZATION constraints provide upper and lower bounds for open facility-plant site combinations. The FAC_ALLOCATION constraints assure that a piece of facility is assigned to only one plant site.

Our intention is to restore the product distribution feature to Model_3 by combining it with Model_2. The corporate production requirement is unbundled into individual customer demands and each open plant site "k" may ship to any destination "l". The integrated model, shown in Figure 6.10, is the result of the following four step procedure:

- S1. Replace Model_2's ORIGIN "k" index set by Model_3's PLANT "k" index set. Redirect all ORIGIN "k" references in Model_2 to PLANT "k" (technique 1).
- S2. Form a new objective function, "TOTAL_COST", by summing the objective functions of the component models (technique 4).
- S3. Define a new constraint, "SUPPLY(k)", that assures that the product shipped from any plant site "k" does not exceed the total finished goods produced at the site on

its assigned facilities. "SUPPLY(k)" is composed of the left-hand side of Model_2's "OUT_BND(k)" constraint set and the expression

$$\text{SUM}(j) \{ \text{ACTIVITY} \} (\text{PROD}(j,k)).$$

S4. Eliminate "QUOTA(k)" and "CORP_DEMAND" from Model_3; eliminate "OUT_BND(k)" and "SUP(k)" from Model_2.

f. Untyped Modeling Language Support of Integrated Modeling

(1) Strengths. The lion's share of the support available to the practitioner of integrated modeling when the model components are written in any modeling language (typed or untyped) is provided by a text editor, not by the modeling language itself. At the modeler's direction, the text editor concatenates model files, eliminates unwanted language statements, moves blocks of text, and finds/replaces character strings designated by the modeler.

After the integrated model text file has been prepared, untyped modeling language translators provide two useful consistency checks. First, they enforce a "define before use" rule. If, for example, the definition of a parameter is eliminated from the model, but the parameter has been left in a constraint by mistake, the modeling language translator will catch the error. Another useful aid is the post-compilation cross-reference listing. The cross-reference listing provides the text line number where each labelled object in the model is declared and used. Objects which are declared but not used can be removed from the integrated model.

(2) Weaknesses. In order for the four model integrations posed above to be meaningful the modeler must insure that certain criteria are satisfied. Consider the composition of the objective functions of Model_1 and Model_2: optimization of their sum only makes sense when both functions are measured in the same units. An additional necessary condition for Examples 2 and 3 to be well-formed is that both Model_1 and Model_2 ship the same quantity of the same thing, that it be measured in the same units,


```

SETS
  FACILITIES ;
  PLANT k ;
  ACTIVITY(j,k) := { SELECT( {FACILITY} , {PLANT} ) };

  ORIGIN k ;
  DESTINATION l ;
  LINK(k,l) := { CROSS( { PLANT } x { DESTINATION } ) };

PARAMETERS
  CORP_DEMAND ;
  PLANT_COST(k) {PLANT} ;
  FAC_LIMIT_LOWER(k) {PLANT} ;
  FAC_LIMIT_UPPER(k) {PLANT} ;

  SUP(k) {PLANT} ;
  DEM(l) {DESTINATION} ;
  COST(k,l) {LINK} ;

  PROD_COST(j,k) {ACTIVITY} ;
  FAC_COST(j,k) {ACTIVITY} ;
  FAC_UTIL_LOWER(j,k) {ACTIVITY} ;
  FAC_UTIL_UPPER(j,k) {ACTIVITY} ;

VARIABLES
  OPEN_PLANT(k) {PLANT} ;
  PROD(j,k) {ACTIVITY} ;
  OPEN_FAC(j,k) {ACTIVITY} ;
  POSITIVE: PROD(j,k);
  BOOLEAN: OPEN_FAC(j,k), OPEN_PLANT(k);

  Y(k,l) {LINK} ;
  POSITIVE: Y(k,l);

FUNCTIONS
  OBJECTIVE := SUM(j,k) {ACTIVITY} (PROD_COST(j,k) * PROD(j,k) + FAC_COST(j,k) *
    OPEN_FAC(j,k)) + SUM(k) {PLANT} (PLANT_COST(k) * OPEN_PLANT(k)) ;

  TOT_$ := SUM(k,l) {LINK} (COST(k,l) * Y(k,l)) ;

  TOTAL_COST := OBJECTIVE + TOT_$ ;

CONSTRAINTS
  QUOTA := SUM(j,k) {ACTIVITY} (PROD(j,k)) =E= CORP_DEMAND ;

  CO_LOCATION(k) {PLANT} := FAC_LIMIT_LOWER(k) * OPEN_PLANT(k)
    =L= SUM(j) {ACTIVITY} (OPEN_FAC(j,k))
    =L= FAC_LIMIT_UPPER(k) * OPEN_PLANT(k) ;

  UTILIZATION(j,k) {ACTIVITY} := FAC_UTIL_LOWER(j,k) * OPEN_FAC(j,k)
    =L= PROD(j,k)
    =L= FAC_UTIL_UPPER(j,k) * OPEN_FAC(j,k) ;

  FAC_ALLOCATION(l) {FACILITIES} := SUM(k) {ACTIVITY} (OPEN_FAC(j,k)) =L= 1 ;

  OUT_BND(k) {PLANT} := SUM(l) {LINK} (Y(k,l)) =L= SUP(k) ;

  SUPPLY(k) := SUM(l) {LINK} (Y(k,l)) =L= SUM(j) {ACTIVITY} (PROD(j,k)) ;

  IN_BND(l) {DESTINATION} := SUM(k) {LINK} (Y(k,l)) =E= DEM(l) ;

```

Figure 6.10 EML Schema for Example 5

and that the temporal setting of both models be the same. (There is a notion of things moving per unit of time in a transportation model. Things arriving at a node, say at a monthly rate, should depart the same node at a monthly rate.) In an untyped modeling language, the modeler is the first and last line of defense for detecting these kinds of errors.

2 . Integrated Modeling In A Typed Language

The mechanics of manipulating the algebraic representations of model components in a typed modeling language are the same as those of an untyped modeling language. The same methods detailed in Section A of this chapter are used and the integrated representation is prepared outside the modeling system environment using a text editor. The important difference between the two mediums is the proportion of responsibility for model verification that the modeling language can assume from the modeler.

In any computer language, the language compiler guarantees that every program it accepts is a legal combination of the operators and labels that comprise its grammar, nothing more. The programmer retains responsibility for the computer representation of the problem and all aspects of the problem which cannot be expressed in the language. Extension of a modeling language grammar through typing increases the proportion of a model which is formal and checkable. In exchange for this adoption of the conventions of a typed language, a modeling language system (translator and type analyzer) assumes more of the error detection responsibilities of the modeler.

To use the error checking abilities of a type language to their full potential, the intentions of the modeler must be expressed within the boundaries formed by the language itself and the types defined for a particular application domain. If the modeler elects to assign a numerical object the universal type instead of a type recognized in the application

domain of the model, the modeler reassumes the responsibility for the dimensional consistency of all constructs that use that object.

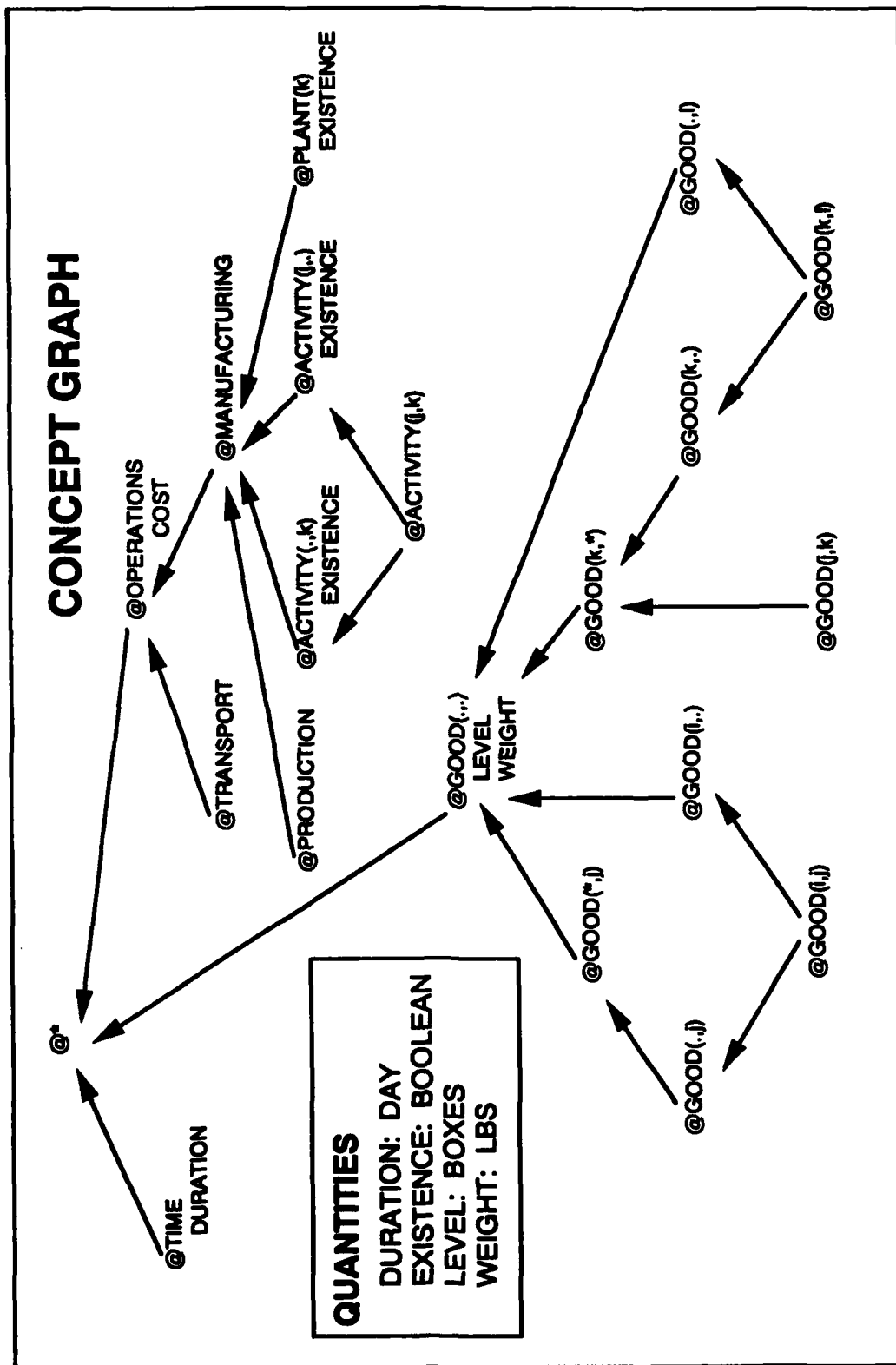
The algebraic interconnections that a modeler introduces between two component models to form an integrated model make assertions about the semantic equivalence of indices, parameters and variables. When a modeling language is typed, a formal basis exists for affirming or denying these assertions. This is done by evaluating each algebraic function and constraint for type correctness. Any replacement, composition or creation which is inconsistent with the definition of the language or the type context generates a type error. If a type error is produced, one or more of the modeler's assertions is false. This testing can be done manually by the modeler or delegated to the modeling language system. Thus, typing provides the modeler an in-depth error defense.

We now consider two examples of this approach. The basis for verifying the interconnections is the type language as defined in Chapter 3 and the concept graph and quantity declaration given in Figure 6.11. We have used this type context to type each of the component models introduced in section B. Typed versions of Model_1, Model_2 and Model_3 are listed in Figures 6.12, 6.13, and 6.14, respectively.

a. The Transshipment Model

The transshipment model that was constructed in section B, Example 3 makes three assertions: one in the form of an index replacement, another in the form of a constraint, and a third in the form of a function. The typed algebraic representation of this model is shown in Figure 6.15.

(1) Index Replacement. When the SINK "j" index set replaced the ORIGIN "k" index set to establish the indexing structure for the integrated model, an assertion was made: the ordering of the elements of the ORIGIN "k" set was not an operative part of component Model_2. That is, Model_2 contained no parameters or variables that were



SETS

SOURCE i ; << *nominal* >>
 SINK j ; << *nominal* >>
 ARC(i,j) := { CROSS({SOURCE} , {SINK}) };

VARIABLES

X(i,j) {ARC}; << *WEIGHT of @GOOD(i,j) / DURATION of @TIME # [100] LBS/DAY # >>*
 POSITIVE: X(i,j);

PARAMETERS

S(i) {SOURCE} ; << *WEIGHT of @GOOD(i,.) / DURATION of @TIME # [100] LBS/DAY # >>*
 D(j) {SINK} ; << *WEIGHT of @GOOD(.,j) / DURATION of @TIME # [100] LBS/DAY # >>*
 C(i,j) {ARC} ; << *COST of @TRANSPORT / WEIGHT of @GOOD(i,j) / DURATION of @TIME) # US_\$ / ([100] LBS/DAY) # >>*

FUNCTIONS

OBJ := SUM (i,j) {ARC} (C(i,j)*X(i,j)) ; << *COST of @TRANSPORT # US_\$ # >>*

CONSTRAINTS

SUPPLY(i) {SOURCE} := SUM (j) {ARC} (X(i,j)) =L= S(i) ; << *WEIGHT of @GOOD(i,.) / DURATION of @TIME # [100] LBS/DAY # >>*
 DEMAND(j) {SINK} := SUM (i) {ARC} (X(i,j)) =E= D(j) ; << *WEIGHT of @GOOD(.,j) / DURATION of @TIME # [100] LBS/DAY # >>*

Figure 6.12 Model_1 (Transportation Model as Typed EML Schema)

identified by lag or lead operations on index values (e.g., "Y(i,j+1)"). When the component models are typed, this assertion can be affirmed or denied by examining the index type declaration of the replaced index set. In this case, "ORIGIN k" is a nominal index set, affirming the substitution.

(2) **Function Composition.** The second assertion in this integrated model is that the objective functions of Model_1 and Model_2 are additive. The propriety of summing the objective functions of Model_1 and Model_2 to form the composite objective function, TOTAL_COST can be checked by determining the type of the sum and

```

SETS
  ORIGIN k ; << nominal >>
  DESTINATION l ; << nominal >>
  LINK(k,l) := { CROSS( {ORIGIN} , {DESTINATION} ) };

VARIABLES
  Y(k,l) {LINK} ; << WEIGHT of @GOOD(k,l) / DURATION of @TIME
                                     # [100] LBS/DAY # >>
  POSITIVE: Y(k,l);

PARAMETERS
  SUP(k) {ORIGIN} ; << WEIGHT of @GOOD(k,.) / DURATION of
                                     @TIME # [100] LBS/DAY # >>
  DEM(l) {DESTINATION} ; << WEIGHT of @GOOD(.,l) / DURATION
                                     of @TIME # [100] LBS/DAY # >>
  COST(k,l) {LINK} ; << COST of @TRANSPORT / ( WEIGHT of @GOOD(k,l)
                                     / DURATION of @TIME ) # US_$ / ( [100] LBS/DAY ) # >>

FUNCTIONS
  TOT_$ := SUM(k,l) {LINK} ( COST(k,l)*Y(k,l) ) ; << COST of @TRANSPORT # US_$ # >>

CONSTRAINTS
  OUT_BND(k) {ORIGIN} := SUM(l) {LINK} ( Y(k,l) ) =L= SUP(k) ;
                                     << WEIGHT of @GOOD(k,.) / DURATION of @TIME # [100] LBS/DAY # >>
  IN_BND(l) {DESTINATION} := SUM(k) {LINK} ( Y(k,l) ) =E= DEM(l) ;
                                     << WEIGHT of @GOOD(.,l) / DURATION of @TIME # [100] LBS/DAY # >>

```

Figure 6.13 Model_2 (Transportation Model as Typed EML Schema)

comparing it to the type statement of TOTAL_COST. Since all three functions have the type << COST of @TRANSPORT # US_\$ # >> the assertion is true.

(3) Constraint Composition. Each MAT_BAL(j) constraint (Figure 6.15) asserts that the decision variables in both model components represent shipment of the same quantity of the same thing, measured in the same units, over the same period of time. Notice that each decision variable in this constraint has a unique type: << WEIGHT of @GOOD(i,j) / DURATION OF TIME # [100] LBS/DAY # >> for X(i,j) and << WEIGHT of @GOOD(j,i) # [100] LBS/DAY # >> for Y(j,i). Also notice that unlike the objective function example just presented, there are no parameters in the material balance

```

SETS
  FACILITIES j ; << nominal >>
  PLANT k ; << nominal >>
  ACTIVITY (j,k) := { SELECT( CROSS( {FACILITY} , {PLANT} ) ) };

PARAMETERS
  FAC_AVAIL(j) {FACILITIES} / 1 / ; << EXISTENCE of @ACTIVITY(j,.) # BOOLEAN # >>
  CORP_DEMAND ; << LEVEL of @GOOD(.,.) # BOXES # >>
  PLANT_COST(k) {PLANT} ; << COST / EXISTENCE of @PLANT(k) # US_$ / BOOLEAN # >>

  FAC_LIMIT_LOWER(k) {PLANT} ; << EXISTENCE of @ACTIVITY(.,k) / EXISTENCE of
                                @PLANT(k) # UNITY # >>

  FAC_LIMIT_UPPER(k) {PLANT} ; << EXISTENCE of @ACTIVITY(.,k) / EXISTENCE of
                                @PLANT(k) # UNITY # >>

  PROD_COST(j,k) {ACTIVITY} ; << COST of @PRODUCTION / LEVEL of @GOOD(j,k)
                                # US_$ / BOXES # >>

  FAC_COST(j,k) {ACTIVITY} ; << COST / EXISTENCE of @ACTIVITY(j,k)
                                # US_$ / BOOLEAN # >>

  FAC_UTIL_LOWER(j,k) {ACTIVITY} ; << LEVEL of @GOOD(j,k) / EXISTENCE of
                                @ACTIVITY(j,k) # BOXES / BOOLEAN # >>

  FAC_UTIL_UPPER(j,k) {ACTIVITY} ; << LEVEL of @GOOD(j,k) / EXISTENCE of
                                @ACTIVITY(j,k) # BOXES / BOOLEAN # >>

VARIABLES
  OPEN_PLANT(k) {PLANT} ; << EXISTENCE of @PLANT(k) # BOOLEAN # >>
  PROD(j,k) {ACTIVITY} ; << LEVEL of @GOOD(j,k) # BOXES # >>
  OPEN_FAC(j,k) {ACTIVITY} ; << EXISTENCE of @ACTIVITY(j,k) # BOOLEAN # >>
  POSITIVE: PROD(j,k);
  BOOLEAN: OPEN_FAC(j,k), OPEN_PLANT(k);

FUNCTIONS
  OBJECTIVE := SUM(j,k) {ACTIVITY} (PROD_COST(j,k) * PROD(j,k) + FAC_COST(j,k) * OPEN_FAC(j,k))
              + SUM(k) {PLANT} (PLANT_COST(k) * OPEN_PLANT(k));
              << COST of @MANUFACTURING # US_$ # >>

CONSTRAINTS

  FAC_ALLOCATION(j) {FACILITIES} := SUM(k) {ACTIVITY} (OPEN_FAC(j,k)) =L= FAC_AVAIL(j) ;
                                << EXISTENCE OF ACTIVITY(j,.) # BOOLEAN # >>

  QUOTA := SUM(j,k) {ACTIVITY} (PROD(j,k)) =E= CORP_DEMAND ;
                                << LEVEL of @GOOD(.,.) # BOXES # >>

  CO_LOCATION(k) {PLANT} := FAC_LIMIT_LOWER(k) * OPEN_PLANT(k)
                            =L= SUM(j) {ACTIVITY} (OPEN_FAC(j,k))
                            =L= FAC_LIMIT_UPPER(k) * OPEN_PLANT(k) ;
                            << EXISTENCE of ACTIVITY(.,k) # BOOLEAN # >>

  UTILIZATION(j,k) {ACTIVITY} := FAC_UTIL_LOWER(j,k) * OPEN_FAC(j,k)
                                =L= PROD(j,k)
                                =L= FAC_UTIL_UPPER(j,k) * OPEN_FAC(j,k) ;
                                << LEVEL of @GOOD(j,k) # BOXES # >>

```

Figure 6.14 Model_3 (Production Model As Typed EML Schema)

SETS

SOURCE I ; << nominal >>
 SINK J ; << nominal >>
 ARC(I,J) := { CROSS({SOURCE}, {SINK}) };

ORIGIN K ; << nominal >>
 DESTINATION L ; << nominal >>
 LINK(I,J) := { CROSS({SINK}, {DESTINATION}) };

VARIABLES

X(I,J) {ARC} ; << WEIGHT of @GOOD(I,J) / DURATION of @TIME # [100] LBS / DAY # >>

Y(J,L) {LINK} ; << WEIGHT of @GOOD(J,L) / DURATION of @TIME # [100] LBS / DAY # >>

POSITIVE: X(I,J), Y(J,L);

PARAMETERS

S(I) {SOURCE} ; << WEIGHT of @GOOD(I,J) / DURATION of @TIME # [100] LBS / DAY # >>
 D(J) {SINK} ; << WEIGHT of @GOOD(J,L) / DURATION of @TIME # [100] LBS / DAY # >>

C(I,J) {ARC} ; << COST of @TRANSPORT / (WEIGHT of @GOOD(I,J) / DURATION of @TIME # US_\$ / ([100] LBS / DAY) # >>

SUP(J) {SINK} ; << WEIGHT of @GOOD(J,L) / DURATION of @TIME # ([100] LBS / DAY) # >>

DEM(L) {DESTINATION} ; << WEIGHT of @GOOD(J,L) / DURATION of @TIME # ([100] LBS / DAY) # >>

COST(J,L) {LINK} ; << COST of @TRANSPORT / (WEIGHT of @GOOD(J,L) / DURATION of @TIME) # US_\$ / ([100] LBS / DAY) # >>

FUNCTIONS

OBJ := SUM(I,J) {ARC} (C(I,J)*X(I,J)) ; << COST of @TRANSPORT # US_\$ # >>

_ TOT_\$:= SUM(J,L) {LINK} (COST(J,L)*Y(J,L)) ; << COST of @TRANSPORT # US_\$ # >>

TOTAL_COST := OBJ + TOT_\$; << COST of @TRANSPORT # US_\$ # >>

CONSTRAINTS

SUPPLY(I) {SOURCE} := SUM(J) {ARC} (X(I,J)) = L = S(I) ; << WEIGHT of @GOOD(I,J) / DURATION of @TIME # [100] LBS / DAY # >>

DEMAND(J) {SINK} := SUM(L) {LINK} (Y(J,L)) = E = D(J) ; << WEIGHT of @GOOD(J,L) / DURATION of @TIME # [100] LBS / DAY # >>

OUT_BND(J) {SINK} := SUM(L) {LINK} (Y(J,L)) = L = SUP(J) ; << WEIGHT of @GOOD(J,L) / DURATION of @TIME # ([100] LBS / DAY) # >>

IN_BND(L) {DESTINATION} := SUM(J) {LINK} (Y(J,L)) = E = DEM(L) ; << WEIGHT of @GOOD(J,L) / DURATION of @TIME # ([100] LBS / DAY) # >>

MAT_BAL(J) {SINK} := SUM (I) {ARC} (X(I,J)) = E = SUM (L) {LINK} (Y(J,L)) ; << WEIGHT of @GOOD(I,J) / DURATION of @TIME # [100] LBS / DAY # >>

Figure 6.15 Transshipment Model as Typed EML Schema

constraint to convert the types of the decision variables into a common type for addition and comparison. This equality is legal within the type context (Figure 6.11) because the concept @GOOD(*,j) has been declared to be a common ancestor of both @GOOD(i,j) and @GOOD(j,i). It allows all X(i,j) decision variables with the same value of the "j" index to be summed and all Y(j,i) decision variables with the same value of the "j" index to be summed. Each sum has the type

<< WEIGHT of @GOOD(*,j) / DURATION of @TIME # [100] LBS/DAY # >>.

The "*" symbol in the concept suffix means that the identity of the suppressed index is not relevant in subsequent determinations of type equivalence. (If the identity of the suppressed index were important, the "." operator would be used instead of the "*" operator). This common generalization of the @GOOD(i,j) and @GOOD(j,i) concepts in the type context affirms that the left-hand side and the right-hand side of each MAT_BAL(j) constraint have equivalent types and are comparable for numerical equality.

b. The Production/Location/Distribution Model

Figure 6.16 is the typed EML schema for the integration of a typed version of the production/location model (Figure 6.13) with a typed transportation model (Figure 6.14). Like the typed transshipment model described in the preceding section, this integrated model also contains interconnections which are assertions about index types (PLANT "k" and ORIGIN "k"), and assertions about the dimensional consistency of a composite objective function (TOTAL_COST) and a set of constraints (SUPPLY(k)). Rather than perform a near duplicate analysis of these interconnections, we draw attention, instead, to two typing artifacts that are unique to the construction of Figure 6.16.

In the component transportation model, the variable Y(k,i) is declared to have the type

<< WEIGHT of @GOOD(k,i) / DURATION of @TIME # [100] LBS/DAY # >>.

SETS

FACILITIES j ; << nominal >>
 PLANT k ; << nominal >>
 ACTIVITY (j,k) := { SELECT(CROSS({FACILITY} , {PLANT})) };

ORIGIN k ; << nominal >>
 DESTINATION l ; << nominal >>
 LINK(k,l) := { CROSS({PLANT} x {DESTINATION}) };

PARAMETERS

FAC_AVAIL(j) {FACILITIES} / 1 / ; << EXISTENCE of ACTIVITY(j,.) # BOOLEAN # >>
 CORP_DEMAND ; << LEVEL of @GOOD(.,.) # BOXES # >>
 PLANT_COST(k) {PLANT} ;
 << COST / EXISTENCE of @PLANT(k) # US_\$ / BOOLEAN # >>
 FAC_LIMIT_LOWER(k) {PLANT} ;
 << EXISTENCE of ACTIVITY(.,k) / EXISTENCE of @PLANT(k) # UNITY # >>
 FAC_LIMIT_UPPER(k) {PLANT} ;
 << EXISTENCE of ACTIVITY(.,k) / EXISTENCE of @PLANT(k) # UNITY # >>
 PROD_COST(j,k) {ACTIVITY} ;
 << COST of @PRODUCTION / LEVEL of @GOOD(j,k) # US_\$ / BOXES # >>
 FAC_COST(j,k) {ACTIVITY} ;
 << COST / EXISTENCE of @ACTIVITY(j,k) # US_\$ / BOOLEAN # >>
 FAC_UTIL_LOWER(j,k) {ACTIVITY} ;
 << LEVEL of @GOOD(j,k) / EXISTENCE of @ACTIVITY(j,k) # BOXES / BOOLEAN # >>
 FAC_UTIL_UPPER(j,k) {ACTIVITY} ;
 << LEVEL of @GOOD(j,k) / EXISTENCE of @ACTIVITY(j,k) # BOXES / BOOLEAN # >>

SUP(k) {PLANT} ;
 << LEVEL of @GOOD(k,.) / DURATION of @TIME # BOXES / DAY # >>
 DEM(l) {DESTINATION} ;
 << LEVEL of @GOOD(.,l) / DURATION of @TIME # BOXES / DAY # >>
 COST(k,l) {LINK} ;
 << COST of @TRANSPORT / (LEVEL of @GOOD(k,l) / DURATION of @TIME)
 # US_\$ / (BOXES / DAY) # >>

ATEMPORAL / 1 / ; << DURATION of @TIME # DAYS # >> ;

VARIABLES

OPEN_PLANT(k) {PLANT} ; << EXISTENCE of @PLANT(k) # BOOLEAN # >>
 PROD(j,k) {ACTIVITY} ; << LEVEL of @GOOD(j,k) # BOXES # >>
 OPEN_FAC(j,k) {ACTIVITY} ; << EXISTENCE of @ACTIVITY(j,k) # BOOLEAN # >>
 POSITIVE: PROD(j,k);
 BOOLEAN: OPEN_FAC(j,k), OPEN_PLANT(k);

Y(k,l) {LINK} ; << LEVEL of @GOOD(k,l) / DURATION of @TIME # BOXES / DAY # >>
 POSITIVE: Y(k,l);

Figure 6.16 Production-Distribution-Location Model (1 of 2 pages)

FUNCTIONS

OBJECTIVE := SUM(j,k) {ACTIVITY} (PROD_COST(j,k) * PROD(j,k) + FAC_COST(j,k) *
OPEN_FAC(j,k)) + SUM(k) {PLANT} (PLANT_COST(k) * OPEN_PLANT(k));
<< COST of @MANUFACTURING # US_\$ # >>

TOT_\$:= SUM(k,l) {LINK} (COST(k,l)*Y(k,l)); << COST of @TRANSPORT # US_\$ # >>

TOTAL_COST := TOT_\$ + OBJECTIVE ; << COST of @OPERATIONS # US_\$ # >>

CONSTRAINTS

FAC_ALLOCATION(l) {FACILITIES} := SUM(k) {ACTIVITY} (OPEN_FAC(j,k))
=L= FAC_AVAIL(l) ;
<< EXISTENCE OF ACTIVITY(j,.) # BOOLEAN # >>

QUOTA := SUM(j,k) {ACTIVITY} (PROD(j,k)) =E= CORP_DEMAND ;
<< LEVEL of @GOOD(.,.) # BOXES # >>

CO_LOCATION(k) {PLANT} := FAC_LIMIT_LOWER(k) * OPEN_PLANT(k)
=L= SUM(j) {ACTIVITY} (OPEN_FAC(j,k))
=L= FAC_LIMIT_UPPER(k)*OPEN_PLANT(k) ;
<< EXISTENCE of ACTIVITY(.,k) # BOOLEAN # >>

UTILIZATION(j,k) {ACTIVITY} := FAC_UTIL_LOWER(j,k) * OPEN_FAC(j,k)
=L= PROD(j,k)
=L= FAC_UTIL_UPPER(j,k)*OPEN_FAC(j,k) ;
<< LEVEL of @GOOD(j,k) # BOXES # >>

OUT_BND(k) {PLANT} := SUM(l) {LINK} (Y(k,l)) =L= SUP(k) ;
<< LEVEL of @GOOD(k,.) / DURATION of @TIME # BOXES / DAY # >>

SUPPLY(k) := SUM(l) {LINK} (Y(k,l)) * ATEMPORAL
=L= SUM(j) {ACTIVITY} (PROD(j,k)) ;
<< LEVEL of @GOOD(*,k) # BOXES # >>

IN_BND(l) {DESTINATION} := SUM(k) {LINK} (Y(k,l)) =E= DEM(l) ;
<< LEVEL of @GOOD(.,l) / DURATION of @TIME # BOXES / DAY # >>

Figure 6.16 Production-Location-Distribution Model (2 of 2 pages)

To integrate the typed transportation model with the typed production/location model it is necessary to change this declaration to

<< LEVEL of @GOOD(k,l) / DURATION of @TIME # BOXES/DAY # >>.

This revision of the quantity of the @GOOD(k,l) dimensional clause and its attendant unit clause is justified by the fact that @GOOD(k,l) inherits both WEIGHT and LEVEL in the concept graph (Figure 6.11). The same revision is made to the type statements of

COST(k,l), SUP(k) and DEM(l). If @GOOD(k,l), @GOOD(k,.), and @GOOD(.,l) did not have the LEVEL quantity, the type analyzer would notify the user of these discrepancies during its syntactic analysis.

The second typing artifact is also related to the type statement of Y(k,l). To make the SUPPLY(k) constraint consistent, it is necessary to reconcile the fact that the type statement of PROD(j,k) does not contain a @TIME dimensional clause. Since this deficiency cannot be corrected automatically, a conversion parameter, "ATEMPORAL", is included in the integrated model to change Y(k,l) from a rate to an instantaneous value.

C. INTEGRATED MODELING WITH LIBRARY UNITS

Although a model may be conceived as several models interconnected, it must have a monolithic physical representation in modeling language to be implemented. Contemporary modeling language compilers have stringent input requirements. A model must satisfy the language grammar. It must contain only unique names and obey the "define before use" principle. Lastly, it must be expressed in a single textfile. In the previous section these standards were achieved manually by the modeler with the assistance of a text editor. The principal disadvantage of this approach is that it is very complex. It requires the modeler to deal with all the index sets, parameters, variables, functions and constraints of both component models simultaneously. In this section we present an alternate representation for an integrated model that preserves the identity of each model component and emphasizes model interconnections while suppressing diversionary detail. The representation is based on an abstraction we call a library unit and a collection of operators for its manipulation.

1. An Introductory Example

Figure 6.17 displays the typed modeling language representation of a transportation model that has been tailored to fulfill a submissive role in an integrated

model. The original model, shown in Figure 6.18, will be referred to as &TRANSPORT; the revision will be referred to as %EASTBOUND. To obtain %EASTBOUND from &TRANSPORT the following changes were made: first, the "j" index of the SINK set was renamed "k"; second, the SOURCE set was replaced with a set of the same type named "TRANSSHIP"; next, the "i" index of the TRANSSHIP set was renamed "j"; and last, the SUPPLY parameter set and the OUTBOUND constraint set were eliminated. (Note: SUPPLY and OUTBOUND were indexed over "j" in {TRANSSHIP} when they were eliminated because TRANSSHIP had replaced SOURCE and "j" had replaced "i" .) The procedural description of how to obtain %EASTBOUND from &TRANSPORT given above can be written concisely in a formal notation. Consider this typed modeling language excerpt, augmented by special purpose operators:

```

SET TRANSSHIP J ; << NOMINAL >>
LIB %EASTBOUND := &TRANSPORT WHERE
    J <- k;
    SOURCE <= TRANSSHIP;
    I <- j;
    ELIM ( SUPPLY(j), OUTBOUND(j) );
END

```

"LIB" is a modeling language keyword that causes an in-line substitution of an exact or a modified copy of an archival model, called a *library unit*. The names of library units are prefaced by the "&" character. Instances of library units are identified by names beginning with the "%" character. The character string "%EASTBOUND := &TRANSPORT" indicates that the left argument is an instance of the right argument.

The differences between the instance and the original are detailed after the keyword "WHERE". If the instance was an exact copy, this keyword would be omitted. The keyword "END" is used to mark the end of the library unit modifications. Three special operators are employed in this description. Operations are applied sequentially; each one assumes that the operations that precede it have been completed. The "<-"

```

<< CONCEPT GRAPH
    @* <-- @OPERATIONS [ COST ]
    @OPERATIONS <-- @TRANSPORT
    @* <-- @TIME [ DURATION ]
    @* <-- @GOOD(j,.) [ WEIGHT ]
    @* <-- @GOOD(.,k) [ WEIGHT ]
    @GOOD(j,.) <-- @GOOD(j,k)
    @GOOD(.,k) <-- @GOOD(j,k) >>

<< UNIT SYSTEMS
    WEIGHT : Avoirdupois_Weight
    COST : US_Currency
    DURATION : Standard_Time >>

SETS
    TRANSSHIP j ; << nominal >>
    SINK k ; << nominal >>
    ARC (j,k) := { CROSS( {TRANSSHIP} , {SINK} ) };

VARIABLES
    FLOW(j,k) {ARC}; << WEIGHT of @GOOD(j,k)/DURATION of @TIME
                                                                # [100] LBS/DAY # >>
    POSITIVE: FLOW(j,k);

PARAMETERS
    DEMAND(k) {SINK}; << WEIGHT of @GOOD(.,k)/DURATION of @TIME
                                                                # [100] LBS/DAY # >>

    COST(j,k) {ARC}; << COST of @TRANSPORT / ( WEIGHT of @GOOD(j,k) /
                                                                DURATION of @TIME ) # US_$ / ( [100] LBS/DAY ) # >>

FUNCTIONS
    OBJECTIVE := SUM(j,k) {ARC} (COST(j,k)*FLOW(j,k)) ; << COST of
                                                                @TRANSPORT # US_$ # >>

CONSTRAINTS
    INBOUND(k) {SINK} := SUM(k) {ARC} (FLOW(j,k) =E= DEMAND(k) ;
    << WEIGHT of @GOOD(.,k) / DURATION of @TIME # [100] LBS/DAY # >>

```

Figure 6.17 %EASTBOUND EML Schema

operator replaces the character string at the head of the arrow with the character string at the tail. The "<=" operator is a type-constrained version of "<-". It has three actions: it erases the definition of the typed object on the left of the operator in the instance; it replaces the character string on the left with the character string on the right; and, it inserts an assertion into the text of the model that the type of the right argument is equivalent to the type of the left argument. This assertion is tested during type verification. If the assertion is false, an error message is generated. In this example, the assertion would be:

```

<< ASSERTION: TYPE(TRANSSHIP j) =?= nominal >>

```

The third operator used in the example is "ELIM()". This operator eliminates the named objects included within its parentheses from the instance. This involves masking object declarations and replacing the names of numerical objects in arithmetic expressions by "0" and "1." The "0" is used when the object is an operand in addition, subtraction or a relational operation. The "1" is used when the object is an operand in multiplication or division.

To preclude any ambiguity that would occur if the LIB keyword was used to create another instance of &TRANSPORT, the names of the objects in %EASTBOUND assumed from &TRANSPORT need to be distinguished. Hence, in all modeling language statements that follow the instantiation of %EASTBOUND, the names of its objects are prefixed with its instance name followed by a period. For example, FLOW(j,k) in Figure 6.17 would be referred to as "%EASTBOUND.FLOW(j,k)."

2 . Library Units

A library unit is a model or fragment of a model that has been kept as a template for building new, perhaps integrated, models. Each library unit has four parts: a type context, a body, a unique name and an interface (e.g., Figure 6.18). The type context contains a concept graph and a measurement system. For example, in Figure 6.18 the quality COST is attributed to the concept @OPERATIONS and measured in US_Currency. The body of a library unit is a typed modeling language representation that can contain index sets, parameters, variables, functions and constraints. The body may be a complete model or a model fragment that contains, for example, data transformations or a collection of constraints. Model fragments, however, are still required to satisfy the "define before use" principle. Each type used in the body can be derived from the concepts, quantities and measurement systems declared in the type context. This, of course, can be verified by applying the type calculus.

```

<< CONCEPT GRAPH
    @*                <-- @OPERATIONS [ COST ]
    @OPERATIONS       <-- @TRANSPORT
    @*                <-- @TIME [ DURATION ]
    @*                <-- @GOOD(i,.) [ WEIGHT ]
    @*                <-- @GOOD(.,j) [ WEIGHT ]
    @GOOD(i,.) <-- @GOOD(i,j)
    @GOOD(.,j) <-- @GOOD(i,j) >>

<< UNIT SYSTEMS
    WEIGHT      : Avoirdupois_Weight
    COST        : US_Currency
    DURATION    : Standard_Time >>

SETS
    SOURCE i ; << nominal >>
    SINK j ; << nominal >>
    ARC (i,j) := { CROSS( {SOURCE} , {SINK} ) };

    VARIABLES
    FLOW(i,j) {ARC}; << WEIGHT of @GOOD(i,j)/DURATION of @TIME
                                                                # [100] LBS/DAY # >>
    POSITIVE: FLOW(i,j);

    PARAMETERS
    SUPPLY(i) {SINK}; << WEIGHT of @GOOD(i,.) / DURATION of @TIME
                                                                # [100] LBS/DAY # >>
    DEMAND(j) {SINK}; << WEIGHT of @GOOD(.,j) / DURATION of @TIME
                                                                # [100] LBS/DAY # >>
    COST(i,j) {ARC}; << COST of @TRANSPORT / ( WEIGHT @GOOD(i,j) /
                                                                DURATION of @TIME ) # US_$ / ( [100] LBS/DAY ) # >>

FUNCTIONS
    OBJECTIVE := SUM(i,j) {ARC} (COST(i,j)*FLOW(i,j));
                                                                << COST of @TRANSPORT # US_$ # >>

CONSTRAINTS
    OUTBOUND(i) {SOURCE} := SUM(j) {ARC} (FLOW(i,j) =L= SUPPLY(i);
                                                                << WEIGHT of @GOOD(i,.) / DURATION of @TIME # [100] LBS/DAY # >>

    INBOUND(j) {SINK} := SUM(i) {ARC} (FLOW(i,j) =L= DEMAND(j);
                                                                << WEIGHT of @GOOD(.,j) / DURATION of @TIME # [100] LBS/DAY # >>

<< INTERFACE
    <=: i, j, @*, @GOOD, @GOOD(i,.), @GOOD(.,j), WEIGHT, @TRANSPORT, COST;
    <=: SINK, SOURCE, ARC, SUPPLY, DEMAND, COST, FLOW, OBJECTIVE,
                                                                OUTBOUND, INBOUND; >>

```

Figure 6.18 &TRANSPORT Library Unit

The type context and body of a library unit are summarized by a unique name and manipulated through two lists of arguments, called an *interface*. One list is headed by the *relabel operator*, "<-", the other by the *replacement operator*, "<=". The presence of a label, index suffix, etc. in a list means that it is a legal left-hand argument for the operator that heads the list. While any character string in the type context or body can appear in the relabel list, only names of typed objects (e.g., variables) can appear in the replacement list. Any index set, parameter, variable, function or constraint in the library unit is a legal argument for the "ELIM(" operator. The contents and organization of the interface are specified by the designer of the library unit to control its usage. When no interface is specified, the only permissible use of the library unit is to copy it verbatim. We envision that a call on a library unit using the "LIB" keyword would be embedded in a model schema as a macro expansion that would replace itself with multiple typed modeling language statements. Before such a model schema would be submitted to a modeling language translator and type analyzer, each "LIB" statement would be replaced by its expanded form. The job of expanding library unit references would be performed by a separate preprocessor. The output of the preprocessor would be a typed modeling language textfile. This full form of the model schema would then be submitted directly to the modeling language translator or, if desired, revised manually by the user before further processing.

In summary, the library unit is intended as means of saving and reusing models. Reuse of models is facilitated through the provision of special operators for relabeling, for replacing typed objects, and for eliminating modeling language objects. These features automate many of the tedious, repetitive symbol manipulations that currently are done to tailor model schemas for new applications.

We caution that these tools can be used improperly. Naive elimination, for example, can make portions of an already validated model dimensionally inconsistent.

Moreover, the ability to replace one index set with another is no guarantee that indexed operations defined over an incumbent set will be implemented in the desired way over a set of smaller or larger dimension. Although errors which result from misuse of a model's interface arguments are detectable during type verification, it is still the modeler's responsibility to understand the library unit and the effects his modifications will have on the consistency of the algebraic and type specification of the model.

3 . Integrated Modeling With Library Units

One obvious advantage of a library unit or any archival model is that it allows modelers to build upon the work of others. The importance of the library unit construct to integrated modeling is its power as an abstraction and as executable documentation. First, by summarizing a model by a unique name and an interface of arguments, the library unit suppresses diversionary detail and emphasizes the modeling constructs that bind component models together. Second, integrated models constructed by combining modeling language statements and library unit invocations provide an executable record of how the full model submitted for model verification was derived from its components. In addition, the use of "%instance_name" prefixes on modeling language identifiers preserves the origin of each construct assumed from a component model.

These advantages are illustrated by reforming two integrated models introduced earlier in this chapter using library units. Figure 6.19 is a transshipment model derived from the &TRANSPORT library unit (Figure 6.18). Figure 6.20 is a production/location/distribution model derived from &TRANSPORT and a production/location library unit, Figure 6.21. See Figures 6.15 and 6.16 for comparison.

```

<< CONCEPT GRAPH
  @*      <- @GOOD(*,j) [ WEIGHT ]
  @GOOD(*,j) <- @GOOD(*,j) >>

LIB %WESTSOURCE :=      &TRANSPORT WHERE
                        SINK <- TRANSSHIP;
                        ELIM( DEMAND(), INBOUND() );
                        END
LIB %EASTBOUND :=      &TRANSPORT WHERE
                        j <- k;
                        i <- j;
                        SOURCE <= %WESTSOURCE.TRANSSHIP;
                        ELIM( SUPPLY(), OUTBOUND() );
                        END

FUNCTIONS
  OBJECTIVE := %WESTSOURCE.OBJECTIVE + %EASTBOUND.OBJECTIVE;
                                                    << COST of @OBJECTIVE # US_$ # >>

CONSTRAINTS
  MAT-BAL() { %WESTSOURCE.TRANSSHIP } := SUM() { %WESTSOURCE.ARC }
  (%WESTSOURCE.FLOW(i,j)) =E= SUM(k) { %EASTBOUND.ARC } (%EASTBOUND.FLOW(j,k));
  << WEIGHT of @GOOD(*,j)/DURATION of @TIME # [100] LBS/DAY # >>

```

Figure 6.19 Transshipment Model Constructed From The &TRANSPORT Library Unit

```

LIB %BISCUITS :=      &PRODUCTION WHERE
                        @GOOD(...) <- @GOOD(*,k);
                        ELIM( CORP_DEMAND, QUOTA );
                        END

SETS
  CUSTOMER l; << nominal >>
  LINK(k,l) := { CROSS( { %BISCUITS.PLANT } , { CUSTOMERS } ) };

LIB %DISTRIB :=      &TRANSPORT WHERE
                        @TRANSPORT <- @DISTRIBUTION;
                        WEIGHT <- LEVEL;
                        [100] POUNDS <- BOXES;
                        i <- k;
                        j <- l;
                        (k,.) <- (*,k);
                        ARC <= LINK;
                        SINK <= CUSTOMER;
                        SOURCE <= %BISCUITS.PLANT;
                        ELIM( SUPPLY(k), OUTBOUND(k) );
                        END

PARAMETERS
  ATEMPORAL / 1 /; << DURATION of @TIME # DAY # >>

FUNCTIONS
  OBJECTIVE := %BISCUITS.OBJECTIVE + %DISTRIB.OBJECTIVE ;
                                                    << COST of @OPERATIONS # US_$ # >>

CONSTRAINTS
  SUPPLY(k) { PLANT } := SUM() { LINK } (%DISTRIB.FLOW(k,l)) * ATEMPORAL
                        =L= SUM () { %BISCUITS.ACTIVITY } (%BISCUITS.PROD(j,k) );
                        << LEVEL of @GOOD(*,k) # BOXES # >>

```

Figure 6.20 Production-Distribution Model Constructed With &PRODUCTION and &TRANSPORT Library Units

```

<< CONCEPT GRAPH
  @* <-- @GOOD(..) [ LEVEL ]
  @GOOD(..) <-- @GOOD(j,k)
  @* <-- @OPERATIONS [ COST ]
  @OPERATIONS <-- @MANUFACTURING
  @MANUFACTURING <-- @PLANT(k) [ EXISTENCE ]
  @MANUFACTURING <-- @PRODUCTION
  @MANUFACTURING <-- @ACTIVITY(..,k) [ EXISTENCE ]
  @MANUFACTURING <-- @ACTIVITY(j,..) [ EXISTENCE ]
  @ACTIVITY(..,k) <-- @ACTIVITY(j,k)
  @ACTIVITY(j,..) <-- @ACTIVITY(j,k) >>

<< UNIT SYSTEMS
  LEVEL : Containers
  COST : US_Currency
  EXISTENCE : Boolean >>

SETS
  FACILITIES j ; << nominal >>
  PLANT k ; << nominal >>
  ACTIVITY (j,k) := { SELECT( CROSS( {FACILITY} , {PLANT} ) ) };

PARAMETERS
  FACILITY_AVAIL(j) {FACILITIES} / 1 ; << EXISTENCE of ACTIVITY(j,..) # BOOLEAN # >>
  CORP_DEMAND ; << LEVEL of @GOOD(..) # BOXES # >>
  PLANT_COST(k) {PLANT} ; << COST / EXISTENCE of @PLANT(k) # US_$ / BOOLEAN # >>
  FAC_LIMIT_LOWER(k) {PLANT} ; << EXISTENCE of @ACTIVITY(..,k) / EXISTENCE of @PLANT(k)
                                                                    # UNITY # >>
  FAC_LIMIT_UPPER(k) {PLANT} ; << EXISTENCE of @ACTIVITY(..,k) / EXISTENCE of @PLANT(k)
                                                                    # UNITY # >>
  PROD_COST(j,k) {ACTIVITY} ; << COST of @PRODUCTION / LEVEL of @GOOD(j,k) # US_$ / BOXES # >>

PARAMETERS
  FAC_COST(j,k) {ACTIVITY} ; << COST / EXISTENCE of @ACTIVITY(j,k) # US_$ / BOOLEAN # >>
  FAC_UTIL_LOWER(j,k) {ACTIVITY} ; << LEVEL of @GOOD(j,k) / EXISTENCE of @ACTIVITY(j,k)
                                                                    # BOXES / BOOLEAN # >>
  FAC_UTIL_UPPER(j,k) {ACTIVITY} ; << LEVEL of @GOOD(j,k) / EXISTENCE of @ACTIVITY(j,k)
                                                                    # BOXES / BOOLEAN # >>

VARIABLES
  PROD(j,k) {ACTIVITY} ; << LEVEL of @GOOD(j,k) # BOXES # >>
  OPEN_FAC(j,k) {ACTIVITY} ; << EXISTENCE of @ACTIVITY(j,k) # BOOLEAN # >>
  OPEN_PLANT(k) {PLANT} ; << EXISTENCE of @PLANT(k) # BOOLEAN # >>
  POSITIVE: PROD(j,k);
  BOOLEAN: OPEN_FAC(j,k), OPEN_PLANT(k);

FUNCTIONS
  OBJECTIVE := SUM(j,k) {ACTIVITY} (PROD_COST(j,k) * PROD(j,k) + FAC_COST(j,k) * OPEN_FAC(j,k)
                                                                    + SUM(k) {PLANT} (PLANT_COST(k) * OPEN_PLANT(k)) ;
                                                                    << COST of @MANUFACTURING # US_$ # >>

CONSTRAINTS
  FAC_ALLOCATION(j) {FACILITIES} := SUM(k) {ACTIVITY} (OPEN_FAC(j,k)) =L= FAC_AVAIL(j) ;
                                                                    << EXISTENCE of ACTIVITY(j,..) # BOOLEAN # >>
  QUOTA := SUM(j,k) {ACTIVITY} (PROD(j,k)) =E= CORP_DEMAND ; << LEVEL of @GOOD(..) # BOXES # >>
  CO_LOCATION(k) {PLANT} := FAC_LIMIT_LOWER(k) * OPEN_PLANT(k)
                                                                    =L= SUM(j) {ACTIVITY} (OPEN_FAC(j,k))
                                                                    =L= FAC_LIMIT_UPPER(k) * OPEN_PLANT(k) ; << EXISTENCE of ACTIVITY(..,k) # BOOLEAN #
                                                                    >>
  UTILIZATION(j,k) {ACTIVITY} := FAC_UTIL_LOWER(j,k) * OPEN_FAC(j,k)
                                                                    =L= PROD(j,k)
                                                                    =L= FAC_UTIL_UPPER(j,k) * OPEN_FAC(j,k) ; << LEVEL of @GOOD(j,k) # BOXES # >>

<< INTERFACE
  <-: @MANUFACTURING, @GOOD, @GOOD(..), @GOOD(j,k), BOXES, LEVEL, "j", "k" ;
  <=: CORP_DEMAND, QUOTA >>

```

Figure 6.21 &PRODUCTION Library Unit

VII. CONCLUSION

We believe that the type calculus for executable modeling languages presented here advances the goal of making model construction, validation and interpretation easier, faster and more reliable. Only the modeler can map real objects into model objects and only the modeler can validate the relationship between reality and model. But, a typed ML provides a context in which there is significant automatic support to check that modeler-defined objects are correctly manipulated. Contemporary MLs without typing have only the universal context of the properties and the laws of algebra: these are too weak to provide any significant verification that the model captures the modeler's intent. Formalizing the modeler's intention in the model schema with typing documents the rationale that underlies the model's algebraic structure. This, in turn, helps model users to interpret model results and to adapt the model in ways that will not violate its tenets.

The inclusion of typing in an ML provides additional automatic support for the logical integration of distinct models. An ML with typing contains information about the meaning of model objects and can automatically check some aspects of the integration that would otherwise be the responsibility of the modeler to do by hand. An ML with typing also offers the opportunity to save and reuse models through the notion of library units and a collection of operations for their manipulation. The importance of the library unit construct to integrated modeling is its power as an abstraction and as executable documentation. First, by summarizing a model by a unique name and an interface of arguments, the library unit suppresses diversionary detail and emphasizes modeling constructs that bind component models together. Second, integrated models constructed by combining modeling language statements and library units provide an executable record of how the full model submitted for validation has been derived from its components.

The type calculus we have developed has limitations. It is adequate for representing functions, equalities and inequalities composed of polynomials with dimensionless exponents. It does not handle transcendental functions or relationships between units which are not multiplicative (e.g., the relationship between the Centigrade and Fahrenheit temperature scales). Another limitation is its reliance on additive homogeneity. While this criterion is endemic to mathematical programming, the predilection to only add, subtract or compare like things to like things is of secondary importance in other fields. For example, in statistics, the merit of an arithmetic predictive equation is measured by how well it explains the tendency of the dependent variable to vary with the independent variable in a systematic fashion. Although the form of the equation may have some real-world basis, such as an exponential form to measure absorption of a chemical in the blood stream, additive homogeneity is an afterthought, accomplished by assigning appropriate dimensions to the numerical constants determined by regression.

In developing typing, some ideas with surprising power have emerged. First, concepts and the calculus to manipulate them are important and should be added to classical dimensional descriptions. Second, typing of objects with multiple indices can be done quite naturally and with as much abstraction as humans employ. Third, some part of the human capacity to effortlessly generalize and specialize typing information can be captured with a rather simple notion of concept graphs. In short, many intuitive ideas that humans use to think about models can be formalized in a very natural way and making this explicit adds to our understanding of models and the modeling process.

APPENDIX A

A PARTIAL GRAMMAR FOR ELEMENTARY MODELING LANGUAGE (EML)

A. IDENTIFIERS AND CONSTANTS

```

<uppercase letter> ::= "A"|"B"|"C"|...|"Z"
<special character> ::= " _ "$"%"
<alphanumeric> ::= <uppercase letter>|<digit>|<special character>
<identifier> ::= <uppercase letter><alphanumeric>*

<digit> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
<unsigned integer> ::= <digit>+ | "POSITION" "(" <index expression> ")"
<signed integer> ::= ["+"|"-"] <unsigned integer>
<decimal number> ::= <unsigned integer> "." <unsigned integer>
<scientific number> ::= <decimal number> "E" ["-"|"+"] <unsigned integer>
<number> ::= <unsigned integer> | <signed integer> | <decimal number>
               | <scientific number>

```

B. SET DEFINITIONS

```

<set definition> ::= "SETS" <declaration list>";"
<declaration list> ::= <declaration>|<declaration> "(" ";" <declaration>)*
<declaration> ::= <set name><index>["/"<element list>"/"]";"
               | <set name> <index tuple> ":"<set rule> ";"
<set name> ::= <identifier>
<index> ::= <lowercase letter>|<index expression>
<index expression> ::= <index>|<index><index lag op> <unsigned integer>
<index lag op> ::= "++"|"--"
<element list> ::= <identifier> "(" ";" <identifier>)*
<index tuple> ::= "(" <index> "(" ";" <index>)* ")"
<set rule> ::= "{" (<set name> | "UNION" "(" <set rule> "," <set rule> ")"
               | "DIFFERENCE" "(" <set rule> "," <set rule> ")"
               | "CROSS" "(" <set rule> "," <set rule> ")"
               | "SELECT" "(" <set rule> [ "(" <index relation list> "]" ] ")"
               | "PROJECTION" "(" <set rule> [ "(" <index> "(" ";" <index>)* "]" ] ")" ) "}"
<element comparison op> ::= "LT"|"LE"|"EQ"|"GE"|"GT"|"NE"
<index relation list> ::= <index relation> ("AND"|"OR" <index relation>)*
<index relation> ::= <index expression><element comparison op><index expression>
               | "NOT" <index expression>
               | "POSITION" "(" <index expression> ")" [ <element comparison op> <unsigned integer> ]

```

C. PARAMETER DEFINITIONS

```

<parameter definition> ::= "PARAMETERS" <parameter list>";"
<parameter list> ::= <parameter declaration>|<parameter declaration>
                        (";"<parameter declaration>)*
<parameter declaration> ::= <parameter name> [{"<set name> "}]["/"<value list>"/"]
<parameter name> ::= <identifier>[<index tuple>]
<value list> ::= <number>(","<number>)*

```

D. VARIABLE DEFINITIONS

```

<variable definition> ::= "VARIABLES" <variable list>";"
<variable list> ::= <variable declaration>|<variable declaration> (";" <variable declaration>)*
<variable declaration> ::= <variable name> [{"<set name>"}]
<variable name> ::= <identifier>[<index tuple>]

```

E. FUNCTION DEFINITIONS

```

<function definition> ::= "FUNCTIONS" <function list>";"
<function list> ::= <function declaration>|<function declaration>(";" <function declaration>)*
<function declaration> ::= <function name> [{"<set name>"}] ":=" <expression>
<function name> ::= <identifier>[<index tuple>]

```

F. EXPRESSIONS

```

<expression> ::= <term>|<unary op><term>|<expression> <binary op><term>
<term> ::= <parameter name>|<variable name>|<function name>|<number>
           | "SUM" <control> "(" <expression> ")"|<index expression>
<control> ::= <index tuple> "{"<set name>"}" [<index relation list>]
<unary op> ::= "-"|"+"
<binary op> ::= "-"|"+"|"**"|"/"|"^"
<logical binary op> ::= "="|"="|"L="|"G="

```

G. CONSTRAINT DEFINITIONS

```

<constraint definition> ::= "CONSTRAINTS" <constraint list>";"
<constraint list> ::= <constraint declaration>|<constraint declaration>(";" <constraint
declaration>)*
<constraint declaration> ::= <constraint name> [{"<set name> "}]"
    ":@"<expression><logical binary op> <expression>
    |<expression> <logical binary op><expression> <logical binary op><expression>
<constraint name> ::= <identifier>[<index tuple>]

```


LIST OF REFERENCES

- Aho, A.V., J.E. Hopcroft and J.D. Ullman <1974>, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- Aho, A.V., and J.D. Ullman <1977>, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.
- Bisschop, J. and A. Meeraus <1982>, "On the development of a General Algebraic Modeling System in a Strategic Planning Environment," *Mathematical Programming Study*, 20 (October 1982), North Holland, Amsterdam, 1-29.
- Bridgeman, W.P. <1937>, *Dimensional Analysis*, The University Press, Yale, New Haven, CT, 1937.
- Brown, G.G., G.W. Graves and M.D. Honczarenko <1987>, "Design and Operation of A Multi-Commodity Production /Distribution System Using Primal Goal Decomposition," *Management Science*, 33, 11 (November 1987), 1469-1480.
- Burger, R.C.<1982>, *MLD: A Language and Data Base for Modeling*, IBM Research Report RC 9639 (#42311), 9 September 1982.
- Clemence, R.D. Jr. <1984>, *LEXICON: A Structured Modeling System for Optimization*, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1984.
- Cohen, M.A. and H.L. Lee <1988>, "Strategic Analysis of Integrated Production-Distribution Systems: Models and Methods," *Operations Research*, 36, 2 (March-April 1988), 216-228.
- Damon, W.W. and R. Schramm <1972>, "A Simultaneous Decision Model for Production, Marketing and Finance," *Management Science*, 19, 2 (October 1972), 161-172.
- Dolk, D.R. and B. Konsynski <1983>, "Model Management in Organizations," *Information and Management*, 9, 1 (June 1983), 35-47.
- Dreiheller, A., M. Moerschbacker, and B. Mohr <1986>, "Programming Pascal with Physical Units," *SIGPLAN Notices*, 21, 12 (December 1986), 114-123.
- Federgruen, A.W.I. and P. Zipkin <1984>, "A Combined Vehicle Routing and Inventory Allocation Problem," *Operations Research*, 32, 5 (September-October 1984), 1019-1037.
- Fourer, R. <1983>, "Modeling Languages Versus Matrix Generators for Linear Programming," *ACM Transactions on Mathematical Software*, 12, 2 (June 1983), 143-183.

Fourer, R., D.M. Gay and B.W. Kernighan <1987>, *AMPL: A Mathematical Programming Language*, Computer Science Technical Report No. 133, AT&T Bell Laboratories, Murray Hill, NJ, January 1987.

Gehani, N.H. <1977>, "Units of Measure as a Data Attribute," *Computer Languages*, 2 (1977), 93-111.

Geoffrion, A.M. <1988>, "SML: A Model Definition Language for Structured Modeling," Western Management Science Institute Working Paper No. 360, University of California Los Angeles, Los Angeles, CA, May 1988.

Goldberg, A. and D. Robson <1983>, *SMALLTALK-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

Hoare, C.A.R. <1973>, *Hints on Programming Language Design*, Stanford University Technical Report STAN-CS-73-C-0435, Palo Alto, CA, December 1973.

House, R.T. <1983>, "A Proposal for an Extended Form of Type Checking of Expressions," *The Computer Journal*, 26, 4 (1983), 366-374.

Karr, M. and D.B. Loveman III <1978>, "Incorporation of Units into Programming Languages," *Communications of the ACM*, 21, 5 (May 1978), 385-391.

Lucas, C., and Mitra, G. <1988>, "Computer-assisted Mathematical Programming (Modeling) System : CAMPS)," *The Computer Journal*, 31, 4 (August 1988), 364-375.

MacLennan, B.J. <1983>, *Principles of Programming Languages: Design, Evaluation and Implementation*, Holt, Rinehart and Winston, New York, NY, 1983.

McInnes, J.M. and W.J. Carleton <1982>, "Theory, Models and Implementation in Financial Management," *Management Science*, 28, 9 (September 1982), 957-978.

Milner, R. <1984>, "A Proposal for Standard ML," Proceedings of the Symposium of LISP and Functional Programming (Austin, TX, August 6-8, 1984), ACM, New York, NY, 184-197, as referenced in Cardelli, L. and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys*, 17, 4 (December 1985), 471-522.

Nachtsheim, T.B. <1987>, *Design and Implementation of a Program Family For Type Evaluation*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1987.

Tremblay, J.P. and P.G. Sorensen <1985>, *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985.

Tutte, W.T. <1948>, "The Dissection of Equilateral Triangles into Equilateral Triangles," *Proceedings of the Cambridge Philosophical Society*, 44, 1948, 463-482.

Ullman, J.D. <1982>, *Principles of Database Systems*, Computer Science Press, Rockville, MD, 1982.

Wiener, R. and R. Sincovec <1984>, *Software Engineering With ADA and Modula-2*, John Wiley & Sons, New York, NY, 1984, 44-50.

BIBLIOGRAPHY

Brachman, R.J., "I Lied About Trees Or, Defaults and Definitions in Knowledge Representation," *The AI Magazine*, 6, 3 (Fall 1985), 80-93.

Brodie, M, J. Mylopoulos, and J. Schmidt (eds.), *On Conceptual Modeling*, Springer-Verlag, Berlin, 1984.

Cardelli, L. and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys*, 17, 4 (December 1985), 471-522.

Sowa, J.F., "Conceptual Graphs for a Database Interface," *IBM Journal of Research and Development*, July 1976, 336-357.

Touretzky, D.S., *The Mathematics of Inheritance Systems*, Morgan Kaufmann, Los Altos, CA, 1986.

INITIAL DISTRIBUTION LIST

- | | | |
|----|--|----|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 0142
Naval Postgraduate School
Monterey, California 93943-5002 | 2 |
| 3. | Chief of Naval Operations (OP-81)
Department of the Navy
Washington, D.C. 20350 | 1 |
| 4. | Office of Naval Research
Mathematics, Code 1111
800 North Quincy Street
Arlington, Virginia 22217 | 1 |
| 5. | Deputy Undersecretary of the Army
for Operations Research
Room 2E621, Pentagon
Washington, D.C. 20310 | 1 |
| 6. | Professor Gordon H. Bradley, Code OR/Bz
Department of Operations Research
Naval Postgraduate School
Monterey, California 93943 | 10 |
| 7. | Professor Gerald G. Brown, Code OR/Bw
Department of Operations Research
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 8. | Professor Daniel R. Dolk, Code AS/Dk
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 9. | Professor Arthur M. Geoffrion
The John E. Anderson
Graduate School of Management
University of California, Los Angeles
Los Angeles, California 90024 | 1 |